UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Persistência e Indexação de Objetos em Sistemas Embarcados.

Mariana Rodrigues

Itajubá, Fevereiro de 2016

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Mariana Rodrigues

Persistência e Indexação de Objetos em Sistemas Embarcados.

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciência e Tecnologia da Computação

Área de Concentração: Hardware e Software Básico

Orientador: Prof. Dr. Enzo Seraphim

Fevereiro de 2016
Itajubá - MG

Agradecimentos

À minha família, pelo apoio incondicional.

Ao Professor Enzo Seraphim, pela paciência e orientação.

Ao Professor Rodrigo Maximiano Antunes de Almeida, pelas sugestões e discussões sobre o tema.

Aos amigos da UNIFEI, sempre dispostos a trocar idéias.

À CAPES, pelo apoio financeiro.

Resumo

Os sistemas embarcados estão presentes em quase todos os equipamentos eletrônicos e é um dos mercados que mais cresce no mundo. Sua rápida evolução de capacidade de armazenamento e processamento abre caminho para o desenvolvimento de aplicações orientadas a dados, as quais possuem requisitos distintos das desenvoldidas para ambiente *desktop*, tais como autonomia, portabilidade e fácil customização. A implementação C++ e a modelagem orientada a objetos do *framework* Object-Injection o torna um ótimo candidato para gerenciamento de dados em sistemas embarcados, já que potencializam a portabilidade e a capacidade de customização do *software*. Este trabalho apresenta a adaptação do *framework* Object-Injection para o sistema FreeRTOS™ com o objetivo de executá-lo em sistemas embarcados de baixo custo. O uso do FreeRTOS™ aumenta a robustez do sistema e permite a construção de aplicações mais complexas com capacidade de temporização (sistemas de tempo real). Experimentos realizados em ambiente simulado e em plataforma embarcada demonstram que a adaptação foi realizada com sucesso, utilizando uma quantidade de recursos computacionais compatível com grande faixa de microcontroladores de baixo custo disponíveis no mercado.

Abstract

Embedded systems are present in almost every electronic equipment and it is one of the most growing markets in the world. The rapid development of theirs storage and processing capabilities make way to data-oriented applications, which have different requisites from those developed to desktop environments, such as autonomy, portability and easy customization. The Object-Injection framework presents itself as a great candidate to manage data in embedded systems because of its C++ implementation and object-oriented modeling, which increases software portability and customization capability. This work present the port of Object-Injection framework to FreeRTOS™ system so that it can be executed in low-cost embedded systems. Using FreeRTOS™ increase system robustness and allows building more complex, real-time applications. Experiments made in both desktop and embedded environments demonstrate the successfulness of the port, using computational resources feasible with many low-cost microcontrollers available in the market.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Programas

Glossário

1 Introdução p. 18

1.1 Motivação . . . . . p. 19

1.2 Objetivo . . . . . p. 19

1.3 Organização do Trabalho . . . . . p. 20

2 Revisão Bibliográfica p. 21

2.1 Sistemas Embarcados . . . . . p. 21

2.1.1 Sistemas Embarcados de Tempo Real . . . . . p. 22

2.1.1.1 O Sistema FreeRTOS™ . . . . . p. 23

2.1.2 Memórias *Flash* . . . . . p. 25

2.2 Banco de Dados para Sistemas Embarcados . . . . . p. 27

2.2.1 Estado da Arte . . . . . p. 28

2.3 O *Framework* Object-Injection . . . . . p. 30

2.3.1 Módulo de Metaclasses . . . . . p. 32

2.3.2 Módulo de Armazenamento . . . . . p. 34

2.3.3 Módulo de Blocos . . . . . p. 35

2.3.4 Módulo de Dispositivos . . . . . p. 37
2.3.5 Organização dos Blocos no Workspace . . . . . p. 39
2.4 Portabilidade de Software . . . . . p. 40
2.4.1 Alinhamento de Memória . . . . . p. 41
2.4.2 Ordenação de Bytes na Memória . . . . . p. 41
2.4.3 Tamanho e Implementação de Tipos Primitivos . . . . . p. 42
2.4.3.1 Sinalização de Caracteres . . . . . p. 42
2.4.3.2 Codificação de Caracteres . . . . . p. 42
2.4.3.3 Sinalização de Tipos Inteiros . . . . . p. 44
2.4.3.4 Tamanho de Tipos Inteiros . . . . . p. 44
2.4.3.5 Representação e Operações de Ponto Flutuante . . . . . p. 45
2.4.4 Utilização de Dados Compostos . . . . . p. 46
2.4.5 Disponibilidade de Recursos Computacionais . . . . . p. 47
2.5 Considerações Finais . . . . . p. 47

3 Framework Object-Injection para Sistemas Embarcados p. 49
3.1 Armazenamento de Objetos Usando Entidades . . . . . p. 49
3.2 Formalização dos Requisitos do Sistema . . . . . p. 52
3.3 Efetivação da Compatibilidade Entre Implementações . . . . . p. 52
3.3.1 Padronização do Tamanho e Sinalização de Tipos Primitivos . . . p. 53
A. Tamanho e Sinalização de Tipos Inteiros . . . . . p. 53
B. Tamanho e Codificação de Caracteres . . . . . p. 54
C. Representação de Tipos de Ponto Flutuante . . . . . p. 55
3.3.2 Padronização da Estrutura de Arquivos . . . . . p. 55
A. Centralização das Operações de Serialização e Desserialização . p. 57
B. Padronização da Ordenação de *Bytes* nas Operações de Serialização e Desserialização . . . . . p. 58

C. Ordenação de *Bytes* para Diferentes Arquiteturas . . . . . p. 58
3.4 Análise das Operações de Escrita em Disco . . . . . p. 60
3.4.1 Avaliação Inicial . . . . . p. 61
3.4.2 Condicionamento da Atualização em Disco . . . . . p. 63
3.4.3 Condicionamento da Atualização do Registro Last Session ID . . p. 65
3.4.4 Comparativo Entre as Implementações . . . . . p. 66
3.5 Integração do *Framework* Object-Injecton ao Sistema FreeRTOS™ . . . . . p. 70
3.5.1 Compilação Conjunta de Códigos nas Linguagens C e C++ . . . . p. 72
3.5.2 Adaptação da Utilização Dinâmica de Memória . . . . . p. 73
3.5.3 Adaptação do Sistema de Arquivos . . . . . p. 75
A. Novo Meio de Armazenamento — A Classe `FATSLFile` . . . . p. 75
B. Inserção de Semáforo para Acesso com Exclusão Mútua . . . . p. 75
C. Criação de Métodos Auxiliares . . . . . p. 77
3.6 Considerações Finais . . . . . p. 78

4 Experimentos p. 79
4.1 Teste de Compatibilidade . . . . . p. 79
4.2 Teste Estimativo . . . . . p. 87
4.3 Teste Funcional . . . . . p. 94
4.3.1 Execução em Ambiente Simulado . . . . . p. 99
4.3.2 Execução em Plataforma Embarcada . . . . . p. 99
4.4 Considerações Finais . . . . . p. 103

5 Conclusão p. 104
5.1 Contribuições . . . . . p. 106
5.2 Trabalhos futuros . . . . . p. 106
5.3 Dificuldades Encontradas . . . . . p. 106

Apêndice A – Programas Utilizados para Automatização de Testes p. 109

A.1 Testes de Escrita em Disco . . . . . p. 109

A.2 Teste de Compatibilidade . . . . . p. 110

A.3 Teste Estimativo . . . . . p. 112

Apêndice B – Resultados Obtidos: Operações de Escrita em Disco p. 114

B.1 Avaliação Inicial . . . . . p. 115

B.2 Condicionamento da Atualização em Disco . . . . . p. 115

B.3 Condicionamento da Atualização do Registro Last Session ID . . . . . p. 115

Referências p. 125

Lista de Figuras

1 Aplicação em camadas utilizando o sistema FreeRTOS™. . . . . p. 23

2 Persistência de entidades e chaves em índices primários e secundários no *framework* Object-Injection. . . . . p. 30

3 Recuperação de entidades a partir de uma chave no *framework* Object-Injection. . . . . p. 31

4 Hierarquia e utilização das interfaces contidas no Módulo de Metaclasses do *framework* Object-Injection (CARVALHO, 2013). . . . . p. 33

5 Hierarquia de serialização e desserialização no Módulo de Metaclasses do *framework* Object-Injection (CARVALHO, 2013). . . . . p. 34

6 Hierarquia do Módulo de Armazenamento do *framework* Object-Injection (CARVALHO, 2013). . . . . p. 35

7 Implementação da classe `Node` no Módulo de Blocos do *framework* Object Injection. . . . . p. 36

8 Hierarquia do Módulo de Blocos do *framework* Object-Injection. . . . . p. 37

9 Cabeçalho (*header*) dos blocos do *framework* Object-Injection (CARVALHO, 2013). . . . . p. 37

10 Hierarquia do Módulo de Dispositivos do *framework* Object-Injection. . . p. 38

11 Constituição de um `workspace` com duas estruturas. . . . . p. 39

12 Estrutura do Bloco Cabeçalho de cada `workspace` do *framework* Object-Injection. . . . . p. 40

13 Estrutura dos Blocos Descritores do *framework* Object-Injection para estruturas de indexação em árvore. . . . . p. 40

14 Padrões de codificação de caracteres definidos na norma Unicode (THE UNICODE CONSORTIUM, 2014). . . . . p. 43

15 Modelagem da estrutura `BTreeEntity` na arquitetura do *framework* Object-Injection. . . . . p. 50

16 Implementação das interfaces ***`Entity`*** e ***`Key`*** para a classe `Student`. . . . . p. 50

17 Tela principal do *ObInject File Dumper*. . . . . p. 56

18 Modelagem da nova classe `Page` e suas derivadas no Módulo de Blocos. . p. 57

19 Modelagem das classes `WriteCounter` e `NumWrites` e sua relação com o Módulo de Dispositivos. . . . . p. 61

20 Avaliação Inicial: Comparativo do tempo total de execução para diferentes tamanhos de páginas. . . . . p. 62

21 Avaliação Inicial: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. . . . . p. 62

22 Metadado `Modified` adicionado à classe `Page`. . . . . p. 63

23 Condicionamento da Atualização em Disco: Comparativo do tempo total de execução para diferentes tamanhos de páginas. . . . . p. 64

24 Condicionamento da Atualização em Disco: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. . . . . p. 64

25 Condicionamento da Atualização do Registro *Last Session ID*: Comparativo do tempo de execução para diferentes tamanhos de páginas. . . . . p. 66

26 Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. . . . . p. 67

27 Comparativo Entre Implementações: Tempo total de execução para operações de persistência e recuperação em arquivo de 4kB de página. . . . . p. 68

28 Comparativo Entre Implementações: Número de operações de escrita em disco na persistência de entidades em arquivo de 4kB de página. . . . . p. 71

29 Frequência de escrita em disco na operação de recuperação de entidades em um arquivo de página de 4kB. . . . . p. 72

30 Classe `CrimeRecord`, suas derivadas e relações com o Módulo de Metaclasses. . . . . p. 81

31 Teste de Compatibilidade: Primeira etapa. . . . . p. 82

32 Teste de Compatibilidade: Segunda etapa. . . . . p. 83

33 Teste Estimativo — Classe `ImageHistogram` e derivadas. . . . . p. 89

34 Teste Estimativo: Resultados obtidos para entidades de 4 dimensões. . . p. 90

35 Teste Estimativo: Resultados obtidos para entidades de 8 dimensões. . . p. 91

36 Teste Estimativo: Resultados obtidos para entidades de 16 dimensões. . . p. 92

37 Teste Estimativo: Resultados obtidos para entidades de 32 dimensões. . . p. 93

38 Classes `Facility` e `Inspection` utilizadas no Teste Funcional. . . . . p. 95

39 Troca de mensagens entre as tarefas `CLI()` e `ObICtr()` na execução do comando `FIND FACILITY INSPECTION ID <id>`. . . . . p. 98

40 Resultado da compilação do conjunto Object-Injection + FreeRTOS™ na plataforma embarcada. . . . . p. 100

41 Teste Funcional: Inicialização da demonstração. . . . . p. 101

42 Teste Funcional: Resultados do sistema após persistência de entidades das classes `Facility` e `Inpection`. . . . . p. 102

43 Teste Funcional: Resultados de comandos de recuperação. . . . . p. 102

44 Teste Funcional: Finalização da demonstração. . . . . p. 103

45 Avaliação Inicial: Tempo de execução. . . . . p. 116

46 Avaliação Inicial: Número de operações de escrita em disco para persistência e recuperação de entidades. . . . . p. 117

47 Avaliação Inicial: Número de operações de escrita em disco para persistência e recuperação de chaves. . . . . p. 118

48 Condicionamento da Atualização em Disco: Tempo de execução. . . . . p. 119

49 Condicionamento da Atualização em Disco: Número de operações de escrita em disco para persistência e recuperação de entidades. . . . . p. 120

50 Condicionamento da Atualização em Disco: Número de operações de escrita em disco para persistência e recuperação de chaves. . . . . p. 121

51 Condicionamento da Atualização do Registro *Last Session ID*: Tempo de execução. . . . . p. 122

52 Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco para persistência e recuperação de entidades. . . . . p. 123

53 Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco para persistência e recuperação de chaves. p. 124

Lista de Tabelas

2 Especificações dos tipos primitivos da linguagem Java (GOSLING et al., 2014). . . . . p. 53

3 Redução no tempo de execução na operação de persistência devido à diminuição da quantidade de operações de escrita em disco. . . . . p. 69

4 Redução no tempo de execução na operação de recuperação devido à diminuição da quantidade de operações de escrita em disco. . . . . p. 70

5 Dados obtidos no Teste de Compatibilidade — Página de 1k. . . . . p. 84

6 Dados obtidos no Teste de Compatibilidade — Página de 2k. . . . . p. 85

7 Dados obtidos no Teste de Compatibilidade — Página de 4k. . . . . p. 86

8 Teste Estimativo — Dimensões e tamanhos de cada entidade utilizada. . p. 88

Lista de Programas

1 Rotinas originais de leitura e escrita de dados tipo `Float`. . . . . p. 51

2 Rotinas modificadas de leitura e escrita de dados tipo `Float`. . . . . p. 51

3 Verificações do arquivo `config.h` para garantir a compilação do *framework* Object-Injection. . . . . p. 52

4 Definição do tipo inteiro customizado `Byte`. . . . . p. 54

5 Verificação do tipo de sinalização de inteiros. . . . . p. 54

6 Teste de representação dos tipos de ponto flutuante. . . . . p. 55

7 Serialização do tipo `Int` pela classe `PushStream`. . . . . p. 56

8 Serialização do tipo `Int` pela classe `Node`. . . . . p. 57

9 Configuração da ordenação de bytes no arquivo `config.h`. . . . . p. 58

10 Leitura de dado `Short` independente da ordenação de *bytes*. . . . . p. 59

11 Leitura de dado `Float` com a configuração de ordenação de *bytes*. . . . . p. 59

12 Modificação do método `flushPage()` na classe `AbstractWorkspace`. . . . p. 63

13 Modificação do construtor de `Session` para armazenamento opicional do campo `LastSessionID`. . . . . p. 65

14 Modificação do arquivo `config.h` para garantir a conformidade da linguagem de programação na utilização do IDE Microsoft Visual Studio. . p. 73

15 Sobrecarga dos operadores de gerenciamento de memória dinâmica no arquivo principal do projeto de demonstração. . . . . p. 74

16 Condicionamento de compilação da classe `FATSLFile`. . . . . p. 76

17 Modificação do método `openSession()` para uso do semáforo na classe `AbstractWorkspace`. . . . . p. 77

18 Modificação do método `close()` para uso do semáforo na classe `Session`. p. 77

19 *Script* utilizado para automatização dos testes de escrita em disco (Seção 3.4). . . . . p. 109

20 *Script* (linguagem R) utilizado para extração dos dados — Teste de Compatibilidade (Seção 4.1). . . . . p. 110

21 *Script* utilizado para execução das repetições — Primeira etapa do Teste de Compatibilidade (Seção 4.1). . . . . p. 111

22 *Script* utilizado para execução das repetições — Primeiro passo da segunda etapa do Teste de Compatibilidade (Seção 4.1). . . . . p. 111

23 *Script* utilizado para execução das repetições (ambiente *desktop*) — Segundo passo da segunda etapa do Teste de Compatibilidade (Seção 4.1). . p. 112

24 *Script* utilizado para execução das repetições (plataforma embarcada) — Segundo passo da segunda etapa do Teste de Compatibilidade (Seção 4.1). p. 112

25 *Script* utilizado para automatização do teste estimativo (Seção 4.2). . . . p. 112

Glossário

API	*Application Programming Interface*
ARM	*Advanced RISC Machine*
ASCII	*American Standard Code for Information Interchange*
BMP	*Basic Multilingual Plane*
CLI	*Command Line Interface*
CPU	*Central Processing Unit*
DOS	*Disk Operating System*
DRAM	*Dynamic Random-Access Memory*
FAT	*File Allocation Table*
FAT SL	*Super Lean FAT File System*
FOP	*Feature Oriented Programming*
HSV	*Hue-Saturation-Value*
I2C	*Inter-Integrated Circuit*
IDE	*Integrated Development Environment*
IEC	*International Electrotechnical Commission*
IEEE	*Institute of Electrical and Electronics Engineers*
IoT	*Internet of Things*
IP	*Internet Protocol*
ISO	*International Organization for Standardization*
JVM	*Java Virtual Machine*
LFS	*Log-Structured File System*
MBR	*Minimum Bounding Rectangle*
MVS	*Microsoft Visual Studio*

P2P	*Peer-to-Peer*
RTOS	*Real-Time Operating System*
RISC	*Reduced Instruction Set Computer*
SGBDs	*Sistema Gerenciador de Banco de Dados*
SPI	*Serial Peripheral Interface*
SPL	*Software Product Line*
SRAM	*Static Random-Access Memory*
SSD	*Solid-State Disk*
SSP	*Synchronous Serial Port*
UART	*Universal Asynchronous Receiver/Transmitter*
UCS	*Universal Coded Character Set*
UDP	*User Datagram Protocol*
USB	*Universal Serial Bus*
UTF	*Unicode Transformation Formats*
UUID	*Universally Unique Identifier*

1 Introdução

Sistemas embarcados estão presentes na maioria dos sistemas computacionais (SAAKE et al., 2009; ROSENMÜLLER et al., 2009a) e é um dos mercados que mais cresce no mundo (QIAN; HARING; CAO, 2009), com previsão de movimentar mais de duzentos bilhões de dólares até o ano de 2020 (Radiant Insights, 2015). Embora sejam praticamente invisíveis para o usuário, estão presentes em quase todos equipamentos modernos à nossa volta, desde um relógio digital a um sistema de navegação para mísseis, passando por telefones celulares, câmeras digitais, brinquedos eletrônicos, aviões, equipamentos médicos e vários outros (JIMNEZ; PALOMERA; COUVERTIER, 2013).

Os sistemas embarcados passaram por uma rápida, contínua e significativa evolução, com o aumento de capacidades de memória, velocidade de processamento e conectividade, o que permitiu o desenvolvimento de sistemas mais complexos a um custo razoável (NORI, 2007). Ao mesmo tempo, a queda contínua do custo das tecnologias de armazenamento (por exemplo, as memórias do tipo *flash*) permite a manipulação de uma quantidade maior de informações, abrindo o caminho para que aplicações orientadas a dados possam ser utilizadas nesses sistemas (NORI, 2007; WHANG et al., 2010).

O desenvolvimento de aplicações gerenciadoras de dados para sistemas embarcados possui muitos desafios inexistentes em sistemas *desktop* (ORTIZ, 2000). Além de endereçar a limitação de recursos computacionais característica, o desenvolvimento deve considerar problemas de interoperabilidade e portabilidade (ORTIZ, 2000) e ser capaz de lidar com novos tipos de dados, não-tradicionais (ABADI et al., 2014). A chamada *Internet das Coisas* (*Internet of Things — IoT*) — uma rede de dispositivos conectados capazes de se comunicar —, trará uma diversidade ainda maior nos dados a serem armazenados (COOPER; JAMES, 2009), em grande parte em dispositivos de recursos limitados. Neste cenário, uma aplicação gerenciadora de dados que possa ser construída em diferentes plataformas de *hardware* é necessária (WHANG et al., 2010).

O campo de sistemas embarcados, embora esteja crescendo rapidamente, tem sido

negligenciado pelos grandes fornecedores de Sistemas Gerenciadores de Banco de Dados (SGBDs) (NORI, 2007). Ainda que existam versões "leves" de SGBDs comerciais, a particularidade de cada dispositivo e a diversidade de aplicações gera uma demanda por aplicações modularizadas que possam ser ajustadas à necessidade de cada sistema (NORI, 2007; WHANG et al., 2010). As pesquisas na área têm procurado preencher esta lacuna, normalmente modularizando algum SGBD comercial ou então desenvolvendo aplicações para dispositivos com recursos muito limitados, como *smart cards*.

1.1 Motivação

O *framework* Object-Injection realiza a indexação e persistência de entidades e chaves utilizando fundamentos da programação orientada a objetos, sem se ater a uma linguagem de programação específica (CARVALHO, 2013), possuindo atualmente implementações nas linguagens Java e C++.

A implementação na linguagem C++ e a abordagem orientada a objetos permitem a consideração do uso do *framework* Object-Injection em sistemas embarcados, uma vez que (1) a linguagem de baixo nível permite o acesso ao *hardware* de forma direta, necessária no campo de sistemas embarcados e desencorajado ou dificultado por linguagens de alto nível (HOOK, 2005); (2) é possível fazer a construção de uma aplicação única, o que permite o uso em ambientes sem sistema operacional; (3) é facilmente customizável, tanto pela linguagem C++ (que permite o uso de diretivas de compilação para configuração de funcionalidades quanto pela fundamentação na Programação Orientada a Objetos (que permite que somente as classes necessárias sejam incluídas no *software*), o que facilita sua adaptação em um campo de aplicações heterogêneas.

1.2 Objetivo

Este trabalho visa a persistência e indexação de objetos utilizando o *framework* Object-Injection em sistemas embarcados sem a utilização de um sistema operacional padrão. Com isso, é obtida uma aplicação capaz de persistir e indexar dados que pode ser utilizada por dispositivos de recursos mais limitados — e portanto de menor custo.

O objetivo final é incluir na implementação C++ do *framework* a capacidade de ser executada sobre o FreeRTOS™ (`www.freertos.org`), um sistema operacional de tempo real que figura entre os mais utilizados nesse mercado atualmente (MERRITT, 2014). O

sistema é implementado na linguagem C, utiliza poucos recursos do dispositivo e sua estrutura em camadas permite fácil adaptação para novas plataformas de *hardware* — atualmente, mais de 30 arquiteturas possuem um código compatível (FreeRTOS™ Features, 2015).

O uso do FreeRTOS™ permite o desenvolvimento de aplicações de tempo real que executem múltiplas tarefas. Além disso, uma vez adaptado, o *framework* pode ser facilmente transferido para qualquer plataforma que possua os requisitos computacionais do conjunto, aumentando o potencial de uso do Object-Injection.

1.3 Organização do Trabalho

Este trabalho apresenta no Capítulo 2 os conceitos estudados para sua realização. São apresentados detalhes sobre o *framework* Object-Injection (Seção 2.3) e conceitos de portabilidade de *software* (Seção 2.4) e sistemas embarcados (Seção 2.1). Em seguida, são analisados aspectos de implementação e desempenho do Object-Injection e apresentadas as ações realizadas para seu aperfeiçoamento e adaptação ao sistema FreeRTOS™ (Capítulo 3). O Capítulo 4 apresenta experimentos executados que comprovam a compatibilidade entre as implementações C++ e Java, além da portabilidade do *software* para outras plataformas e sua bem sucedida adaptação para o sistema FreeRTOS™. Por fim, são apresentadas no Capítulo 5 conclusões e propostas de trabalhos futuros.

2 Revisão Bibliográfica

2.1 Sistemas Embarcados

Um sistema *embutido* ou *embarcado* é um conjunto de *hardware* e *software* fortemente integrados que, diferentemente de sistemas computacionais de uso geral — nos quais o usuário pode instalar vários programas para realizar tarefas diversas —, é projetado para realizar uma função específica. É esperado que o sistema seja autônomo, funcionando sem modificações e com pouca ou nenhuma intervenção humana por um grande período de tempo, geralmente realizando a mesma tarefa repetidamente (QIAN; HARING; CAO, 2009; JIMNEZ; PALOMERA; COUVERTIER, 2013).

O *hardware* de um sistema embarcado deve essencialmente possuir três componentes: (1) uma CPU (*Central Processing Unit*— Unidade de Processamento Central), responsável pela execução do *software*; (2) memórias para armazenamento do programa e de dados; e (3) interfaces de entrada e saída que conectam os outros componentes ao mundo externo.

O *software* do sistema embarcado, frequentemente também chamado de *firmware*, é projetado especificamente para a aplicação, executando-a repetidamente. Uma vez carregado, o *firmware* não sofre modificações, e é esperado que permaneça em funcionamento sem intervenção humana.

A estratégia de implementação do *software* pode variar dependo da complexidade do sistema. Sistemas mais simples podem ser desenvolvidos através da arquitetura de *loop* único (o programa principal é um *loop* infinito e as tarefas são executadas invocando métodos), enquanto sistemas mais complexos são organizados sobre um sistema operacional, este também podendo ser mais simples ou complexo (QIAN; HARING; CAO, 2009; JIMNEZ; PALOMERA; COUVERTIER, 2013).

O desenvolvimento de sistemas embarcados é marcado pelo desafio de superar várias restrições. Além da atenção a parâmetros como o custo e o tempo despendido no

desenvolvimento, são citadas as seguintes restrições (QIAN; HARING; CAO, 2009; JIMNEZ; PALOMERA; COUVERTIER, 2013):

- **Capacidade de processamento:** grande parte dos sistemas embarcados utilizados ainda são sistemas com processadores de pouca potência, com palavras de 8 ou 16 *bits* e velocidades de processamento mais baixas.
- **Disponibilidade de memória:** tipicamente, a disponibilidade de memória de sistemas embarcados é limitada, exigindo em alguns casos grande zelo por parte do desenvolvedor para o funcionamento adequado do sistema.
- **Consumo de energia:** em sistemas embarcados autônomos, especialmente em sistemas nos quais é esperado um tempo de operação prolongado, o consumo de energia e a disponibilidade de bateria têm papel importante em ambos os projetos de *hardware* e *software*.
- **Confiabilidade:** muitas vezes, sistemas embarcados são utilizados em situações de risco à vida ou em ambientes hostis, tornando a confiabilidade uma característica importante. Os sistemas devem ser capazes de se recuperar de situações de erro, continuando a operar normalmente ou então implementando algum conjunto de ações.

2.1.1 Sistemas Embarcados de Tempo Real

Um sistema é dito de *tempo real* (*real-time system*) quando existem restrições de tempo na sua execução, isso é, o sistema possui a habilidade de executar tarefas de forma temporizada. Esses sistemas são classificados de acordo com as consequências de um eventual atraso na resposta, podendo ser considerados *críticos* (*hard real-time systems*) e *não-críticos* (*soft real-time systems*). Em sistemas críticos, um atraso na resposta pode ocasionar uma falha geral do sistema, resultando em danos à propriedade ou mesmo à vida; já em sistemas não-críticos, um atraso na resposta afeta o desempenho mas não compromete a operação do sistema como um todo (QIAN; HARING; CAO, 2009; BARR, 1999).

Em sistemas mais complexos, o desenvolvimento é feito sobre um *Sistema Operacional de Tempo Real* (*RTOS — Real-Time Operating System*) que dê suporte às requisições de tempo do projeto. Nesta configuração, a aplicação é dividida em *tarefas* (*tasks*) responsáveis por uma parte da funcionalidade. Elas são controladas e gerenciadas pelo núcleo do sistema operacional, comumente denominado *kernel*, que também tem como função gerenciar o acesso a serviços e recursos de *hardware* (JIMNEZ; PALOMERA; COUVERTIER,

2013).

2.1.1.1 O Sistema FreeRTOS™

O sistema FreeRTOS™ (`www.freertos.org`) é um dos sistemas operacionais de tempo real mais utilizados no mercado de sistemas embarcados atualmente (MERRITT, 2014). O *software*, implementado em linguagem C, é organizado de forma a construir uma aplicação em camadas, conforme ilustrado pela Figura 1. A aplicação do usuário implementa as tarefas necessárias (*tasks*) e as inclui no *kernel* por meio da API fornecida pelo sistema. O *kernel* inclui um escalonador de execução, métodos de gerenciamento de memória *heap* e outras funcionalidades como filas (*queues*) para comunicação e sincronização de tarefas, semáforos para controle de acesso a dispositivos de *hardware*, rotinas de temporização e ainda ferramentas de análise, como rastreamento (*tracing*) de operações e monitoramento da pilha de *Stack*. Além disso, o sistema oferece extensões, cada qual com sua API e totalmente integradas ao *kernel*, que implementam funcionalidades adicionais, tais como sistemas de arquivo FAT, suporte a comandos, conexão P2P e pilha UDP/IP. O acesso aos recursos de *hardware* é realizado por meio de métodos pré-definidos na chamada *Camada Portátil*. Esses métodos, invocados pelo sistema, devem ser implementados para cada plataforma de *hardware* (FreeRTOS™ Features, 2015).

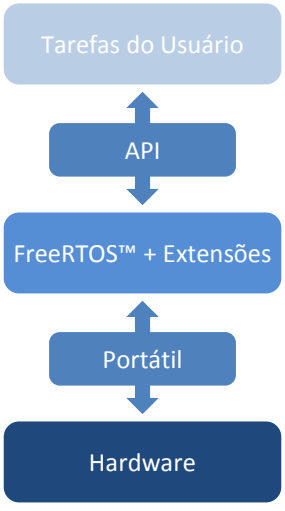


Figura 1: Aplicação em camadas utilizando o sistema FreeRTOS™.

O sistema FreeRTOS™ é customizado e configurado pelo arquivo `FreeRTOSConfig.h`, que deve ser inserido em cada aplicação. Nele, é possível configurar características operacionais (como a preemptividade, a quantidade de memória *heap* disponível e o intervalo de

tempo de execução de cada tarefa) e habilitar/desabilitar funcionalidades como semáforos e supervisão da pilha de *stack*.

O gerenciamento de memória *heap* do sistema FreeRTOS™ aloca memória cada vez que um de seus objetos (como por exemplo uma tarefa ou um semáforo) é criado sem, no entanto, utilizar os métodos `malloc()` e `free()` da biblioteca padrão da linguagem C. Segundo seus desenvolvedores (FreeRTOS™ — Memory Management, 2015), a chamada direta dos métodos da biblioteca padrão poderia causar problemas, uma vez que (1) nem sempre estão disponíveis para sistemas embarcados; (2) ocupam muito espaço na memória de programa; (3) não são *thread-safe* e nem determinísticas. Por isso, todas as alocações/liberações de memória são feitas, respectivamente, pelos métodos `pvPortMalloc()` e `vPortFree()`, pertencentes à camada portátil, podendo ser implementadas de acordo com a arquitetura utilizada. Atualmente são fornecidas ao usuário cinco implementações diferentes para gerenciamento da memória *heap*. O usuário também pode fornecer uma implementação própria ou ainda utilizar duas implementações distintas (FreeRTOS™ — Memory Management, 2015).

A extensão FAT SL (*Super Lean FAT File System*) é uma extensão integrada ao sistema FreeRTOS™ que permite o uso do sistema de arquivos FAT (*File Allocation Table*). A implementação é compatível com o sistema DOS e disponibiliza em sua API tanto métodos para gerenciamento de disco (como inicialização, formatação, criação e gerenciamento de diretórios) quanto para manipulação de arquivos (como leitura e escrita de caracteres e páginas). A interface da extensão com o *hardware* deve ser disponibilizada pela camada portátil por meio de métodos pré-definidos utilizados pela extensão. Possui licença dual, e sua versão gratuita tem a limitação de manipular apenas um arquivo por vez (Super Lean FAT File System, 2015).

A extensão CLI (*Command Line Interface*) fornece uma estrutura eficiente para que a aplicação processe comandos. O desenvolvedor deve (1) prover um método que implemente o comando; (2) associar o método a uma *string* de comando através de uma estrutura `CLI_Command_Definition_t` e (3) registrar o comando através do método `FreeRTOS_CLIRegisterCommand()`. É necessário que a aplicação forneça a interface com o usuário para a recepção dos comandos (por exemplo, pela porta serial ou por uma conexão de *socket*), invocando o processador de comandos da extensão quando necessário (FreeRTOS™+CLI, 2015).

2.1.2 Memórias Flash

Sistemas embarcados geralmente utilizam memórias do tipo *flash* como meio de armazenamento secundário (BOUKHOBZA et al., 2014) devido à características como a não-volatilidade[1], baixo custo e alta densidade (MICHELONI; MARELLI; COMMODARO, 2010).

Os principais tipos de memória *flash* são classificados pelo tipo de porta lógica utilizada em sua construção. Memórias do tipo NOR conseguem acessar dados de forma individualizada e são mais confiáveis; contudo, possuem menor densidade e custo mais alto. Normalmente, são utilizadas para armazenamento de códigos como substitutas de memória DRAM. Memórias do tipo NAND são endereçadas em bloco e possuem maior densidade, sendo utilizadas em mídias removíveis e como armazenamento secundário de dispositivos portáteis (BOUKHOBZA et al., 2014; WANG et al., 2015).

Memórias do tipo NAND são a tecnologia mais utilizada no mercado de cartões de memória (IACULO et al., 2008). Suas características principais são citadas a seguir.

- **Ausência de Elementos Mecânicos:** Os cartões e discos construídos com memória *flash* são dispositivos puramente eletrônicos e, por não possuírem a latência mecânica dos discos comuns, são resistentes à condições ambientais adversas e podem ser inicializados mais rapidamente (KOLTSIDAS; VIGLAS, 2011).
- **Eficiência Energética:** Como são dispositivos puramente eletrônicos, a memória *flash* consume pouca energia e possui baixa dissipação térmica, tornando-as ideais para dispositivos móveis (KOLTSIDAS; VIGLAS, 2011).
- **Assimetria das Operações de I/O:** Devido à características elétricas, as operações de leitura são mais rápidas do que as operações de escrita (KOLTSIDAS; VIGLAS, 2011). Assim, é importante que o número de escritas na memória seja minimizado para aumentar a eficiência do sistema (WANG et al., 2015).
- **Limitação das Operações de Acesso:** As memórias do tipo NAND não permitem o acesso de *bytes* individualmente; as operações de escrita e leitura devem ser feitas em páginas (tipicamente de 512 *bytes*). Ainda, as operações de escrita só conseguem mudar o valor lógico de um bit de 1 para 0, fazendo com que a página tenha que ser apagada para garantir o valor inicial 1. Não é possível, entretanto, apagar uma página individualmente — as operações só podem ser feitas em blocos, compostos de várias páginas. Apagar um bloco de memória é computacionalmente mais custoso do que as operações de escrita e leitura (KOLTSIDAS; VIGLAS, 2011; GAL; TOLEDO, 2005).

[1] Memórias não voláteis retêm os dados quando a alimentação é removida.

- **Vida Útil Limitada:** As páginas de memória podem ser escritas um número limitado de vezes, depois do qual são consideradas desgastadas (GAL; TOLEDO, 2005). Por isso, a diminuição do número de vezes em que a memória é escrita também contribui para o aumento da vida útil do dispositivo (WANG et al., 2015).

Para lidar com as limitações citadas acima, a memória NAND é gerenciada por um controlador. Este pode ser implementado em *hardware* — caso de cartões de memória, *pen drives* ou discos de estado sólido (*Solid State Disks* — SSDs) — ou por *software* — caso dos sistemas operacionais (BOUKHOBZA et al., 2014). As principais funções do controlador são:

- **Mapeamento entre as memórias lógica e física:** quando há a requisição de atualização de uma página da memória, o controlador escreve os dados em um bloco físico disponível (processo chamado modificação *out-of-place*) e atualiza uma tabela de mapeamento, sinalizando o bloco de dados original como inválido (BOUKHOBZA et al., 2014; GHILARDELLI; CORNO, 2010).
- **Nivelamento de uso da memória (*Wear leveling*):** Uma vez que as memórias possuem um limite do número de escritas possíveis em cada página, o controlador deve distribuir essas operações por toda a superfície da memória, impedindo que algumas páginas se desgastem enquanto outras permanecem subutilizadas (GHILARDELLI; CORNO, 2010)
- **Recuperação de páginas inválidas (*Garbage collecting*):** Conforme o número de páginas inválidas aumenta, é necessário que elas sejam recuperadas para que haja espaço disponível para novas operações de escrita. Como só é possível apagar um bloco — este composto de várias páginas —, é necessário transferir as páginas válidas daquele bloco para uma nova localização antes de apagá-lo. Este conjunto de operações é chamado de *Garbage Collection* (WANG et al., 2015). Este algoritmo deve ser executado periodicamente — esporádico o suficiente para que os setores apagados tenham o mínimo de blocos válidos e frequente o suficiente para que o programa não deixe de funcionar por falta de memória —, visando o equilíbrio entre uma alta eficiência e alta performance (GHILARDELLI; CORNO, 2010).

No caso específico de meios externos de armazenamento (cartões de memória e *pen drives*), o controlador é responsável ainda pela comunicação com o mundo externo (IACULO et al., 2008), que deve ser feita de forma eficiente, sem comprometer a velocidade de transferência ou a integridade e confiabilidade dos dados (MICHELONI; MARELLI; COMMODARO, 2010).

2.2 Banco de Dados para Sistemas Embarcados

O desenvolvimento de processamento, memória e conectividade da última década permitiu a popularização e aperfeiçoamento de dispositivos móveis como sensores, aparelhos de celular e navegadores GPS (NORI, 2007). Com a popularização destes dispositivos, surgiu a demanda por aplicações que possuam capacidade de gerenciamento de dados, ou seja, que contivessem um *banco de dados* (WHANG et al., 2010).

Um *Banco de Dados* pode ser definido como "uma coleção de dados relacionados" (ELMASRI; NAVATHE, 2011). Entre os tipos mais comuns de banco de dados, são destacados os (1) *relacionais*, que armazenam as informações em tabelas inter-relacionadas, (2) *hierárquicos*, que armazenam as informações em uma estrutura de árvore, e (3) *orientados a objetos*, que trabalham com objetos mapeados de uma linguagem orientada a objetos, como C++ ou Java (YADAVA, 2007).

Geralmente, o banco de dados é acessado e/ou gerenciado por um programa ou coleção de programas, formando o chamado *SGBD* — Sistema Gerenciador de Banco de Dados (SILBERSCHATZ; KORTH; SUDARSHAN, 2011).

Os SGBD's tradicionais, entretanto, não se mostram adequados para utilização em sistemas embarcados, dadas as restrições características destes sistemas, o amplo espectro de aplicações e a heterogeneidade dos dados a serem armazenados (SAAKE et al., 2009). Assim, o desenvolvimento de um sistema para gerenciamento de dados em sistemas embarcados se mostra muito mais desafiador do que seus correspondentes em sistemas *desktop* (NORI, 2007; ORTIZ, 2000). Algumas características desejadas dos bancos de dados para sistemas embarcados são discutidas a seguir.

- **Autonomia e longevidade:** como qualquer aplicação em sistemas embarcados, os bancos de dados são construídos com um propósito específico e é esperado que funcionem com pouca ou nenhuma interação humana por um grande período de tempo. Assim, o banco de dados deve operar de forma transparente para o usuário, realizando as tarefas de gerenciamento automaticamente (NORI, 2007; WHANG et al., 2010). A capacidade de gerenciamento remoto, entretanto, pode ser relevante em um cenário onde os dispositivos são configurados e gerenciados de forma centralizada (NORI, 2007).
- **Feito sob medida:** devido às restrições de *hardware* características dos sistemas embarcados (como baixa disponibilidade de memória, menor capacidade de processamento e, em vários casos, baixa disponibilidade de bateria), o banco de dados deve

ser ajustado para atender exatamente às necessidades da aplicação, impactando minimamente seu tamanho (*footprint*) e desempenho (NORI, 2007). As funcionalidades do banco de dados são construídas como parte da aplicação, resultando em um produto único (KANG et al., 2009). Também deve ser buscado um equilíbrio entre o desempenho e o uso de recursos, levando em consideração as limitações do *hardware* utilizado (NORI, 2007).

- **Modularizado:** as funcionalidades de banco de dados exigidas em sistemas embarcados variam em praticamente todas as aplicações, sendo que na maioria das vezes é necessária somente uma pequena parte das funcionalidades disponíveis em um SGBD comercial (NORI, 2007; SAAKE et al., 2009). Assim, é desejável um banco de dados com funcionalidades modulares. Neste caso, todas as funcionalidades são implementadas previamente, e somente o necessário é incluído na aplicação (WHANG et al., 2010). Bancos de dados modularizados também possuem maior potencial para se adaptarem a novas tecnologias (NORI, 2007).
- **Multiplataforma:** a construção em conjunto com a aplicação requer um gerenciamento de dados extremamente portátil para que seja possível sua utilização em outras plataformas e sistemas operacionais. Uma das medidas para aumentar a portabilidade é a utilização do banco de dados em arquivo único (NORI, 2007). Alguns desenvolvedores utilizam a linguagem Java para assegurar a portabilidade, descartando a necessidade de adaptação do *software* para diferentes plataformas. Tal medida, entretanto, exige mais recursos computacionais para a execução da JVM, limitando o espectro de dispositivos que podem executar o *software* (ORTIZ, 2000).
- **Capacidade de sincronização:** em várias aplicações distribuídas, não é viável manter os nós sempre conectados. Assim, o banco de dados deve ser capaz de sincronizar a sua porção dos dados com uma fonte central e ainda consolidar dados recebidos de outras fontes (WHANG et al., 2010).
- **Seguro:** geralmente, a segurança não é um problema quando o banco de dados não se comunica com o mundo exterior. Entretanto, medidas de segurança devem ser implementadas para proteger dados sigilosos (WHANG et al., 2010; ORTIZ, 2000). O *software* também não deve armazenar nenhum tipo de código para torná-lo mais seguro e portátil (NORI, 2007).

2.2.1 Estado da Arte

As pesquisas na área de banco de dados para sistemas embarcados incluem protótipos de pesquisa e produtos comerciais. Neste último caso, o desenvolvimento foi focado

principalmente na diminuição de recursos computacionais, sem haver adaptações para o suporte de novos tipos de dados ou para novas tecnologias de armazenamento, como a memória do tipo *flash* (WHANG et al., 2010). São citados como exemplo o IBM DB2 Everyplace (KARLSSON et al., 2001), Microsoft SQL Server Compact (Microsoft SQL Server Compact, 2015), Oracle 10g Lite (Oracle Database Lite Client 10g, 2014) e Oracle Berkeley DB (Oracle Berkeley DB, 2014). Todos os exemplos supracitados têm como instrução de uso a instalação em um sistema operacional.

Em relação a protótipos de pesquisa, são observados dois tipos de aplicação predominantes: (1) bancos de dados para aplicações com memória ultra limitada, como *smart cards* e nós de redes de sensores sem fio, e (2) estratégias de modularização de banco de dados já existentes.

Na primeira categoria, Pucheral et al. (2001) apresentaram o PicoDBMS, um banco de dados voltado para *smart cards* que visa reduzir o tamanho do banco de arquivos e realizar requisições sem utilizar memória RAM. Madden et al. (2005) apresentaram o TinyDB, um sistema que processa as requisições ao banco de dados de maneira distribuída em uma rede de sensores sem fio. Também no contexto de redes de sensores sem fio, foram apresentados os sistemas LittleD (DOUGLAS; LAWRENCE, 2014) e Antelope (TSIFTES; DUNKELS, 2011) que permitem, respectivamente, a execução de requisições e o uso de bancos e índices em tempo de execução, e o FAME-DBMS (SIEGMUND et al., 2009), confeccionado a partir de linhas de produção de software (*Software Product Lines* — SPL). Nath e Kansal (2007) apresentaram o FlashDB, um banco de dados para redes de sensores sem fio que, além de incluir funcionalidades de indexação e compilação de requisições, implementou um controlador para gerenciar o armazenamento em memórias do tipo *flash*. Também pensando nas particularidades das memórias *flash*, Kim et al. (2006) usaram na implementação do LGeDBMS um sistema de arquivos baseado em *logs* (LFS — *Log-structured file system*) para diminuir a quantidade de operações de escrita na memória.

Na segunda categoria, são citados os trabalhos de Rosenmüller et al. (2009b), que utiliza LFS e a programação orientada a funcionalidades (*Feature-Oriented Programming — FOP*) para realizar a modularização do FAME-DBMS; Saake et al. (2009) e Rosenmüller et al. (2009a) utilizam a mesma técnica para a modularização do Berkeley DB — que também foi modularizado por Rosenmüller et al. (2009b).

Outras frentes de pesquisa também estão presentes. Kang et al. (2009), por exemplo, apresentam o QeDB, um banco de dados voltado para sistemas de tempo real com suporte a QoS, cuja implementação foi feita expandindo o popular Berkeley DB (KANG et al.,

2009, 2012).

Na categoria de *software* livre, são citados o SQLite (SQLite, 2014), um banco relacional implementado por meio de uma biblioteca na linguagem C que não precisa de instalação, e o Firebird (Firebird, 2014) um banco também relacional compatível com Linux, Windows e várias plataformas Unix que oferece ao usuário uma API em linguagem C++. Ambos os produtos aceitam comandos da linguagem SQL.

2.3 O Framework Object-Injection

O *framework* Object-Injection realiza a indexação e persistência de entidades (ou objetos) e chaves (ou atributos) por meio de interfaces pré-definidas, baseando-se nos conceitos de *índices primários* e *secundários* (CARVALHO, 2013).

Os índices primários são determinados pela implementação da interface ***Entity*** (Seção 2.3.1) e realizam a indexação de entidades através de um Identificador Único Universal (*Universally Unique Identifier* — UUID), um número inteiro não sinalizado representado por 32 dígitos hexadecimais atualmente especificado pela norma ISO 11578 (CARVALHO, 2013). O UUID de cada entidade permite a sua recuperação independente do dispositivo de armazenamento (CARVALHO, 2013).

Os índices secundários são definidos pela implementação das sub-interfaces de ***Key*** (Seção 2.3.1) e realizam a indexação de chaves em estruturas que possuem relações de ordenação (indexadas em uma Árvore B (ELMASRI; NAVATHE, 2011)), medidas (indexadas em uma Árvore M (CIACCIA; PATELLA; ZEZULA, 1997)) e espaço (indexadas em uma Árvore R (GUTTMAN, 1984)). Os atributos da entidade que definem a chave de indexação são replicados nos índices secundários, assim como o seu UUID (CARVALHO, 2013).

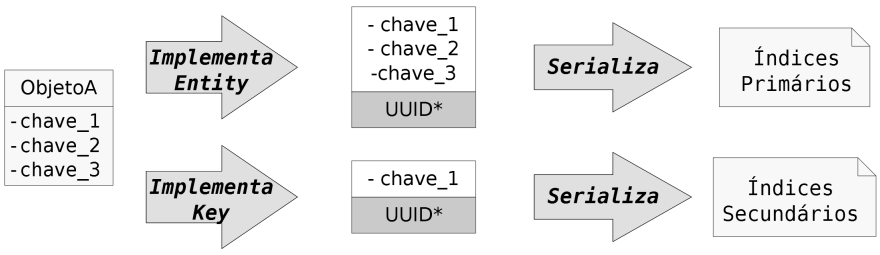


Figura 2: Persistência de entidades e chaves em índices primários e secundários no *framework* Object-Injection. Cada entidade recebe um identificador único (UUID), que é replicado com sua chave nos índices secundários.

A Figura 2 apresenta uma visão geral da operação de persistência. Neste exemplo, deseja-se persistir a instância da entidade `ObjetoA` e também indexá-la pelo valor de uma de suas chaves (`chave_1`). Para ser persistida, a entidade implementa (através de cadeia de herança) a interface ***`Entity`*** e recebe um identificador único (UUID). Os dados da entidade (o valor de suas chaves) e seu UUID são serializados e então armazenados em índices primários. A implementação da interface ***`Key`*** realiza a replicação da chave utilizada na indexação e do UUID da entidade, os quais são serializados e armazenados em índices secundários.

O armazenamento de entidades e chaves em índices primários e secundários permite a recuperação de uma entidade a partir do valor de uma chave, conforme ilustrado pela Figura 3. Neste exemplo, é desejado recuperar a entidade cuja chave indexada possua um valor determinado (por exemplo, `chave_1 = 10`). Para a recuperação da entidade, o valor da chave é buscado nos índices secundários e, caso encontrado, recuperado em conjunto com um UUID. O UUID pode então ser utilizado para a recuperação integral da entidade através de uma busca nos índices primários.

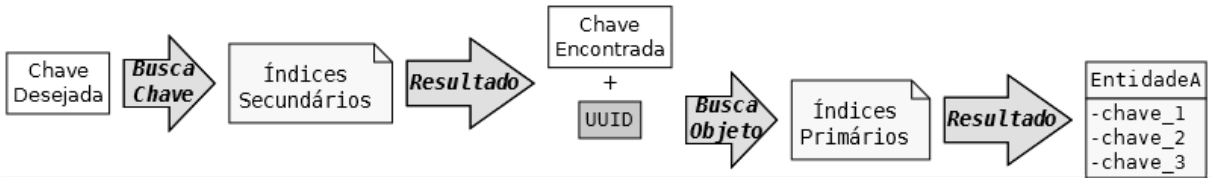


Figura 3: Recuperação de entidades a partir de uma chave no *framework* Object-Injection. A chave desejada é recuperada dos índices secundários em conjunto com um UUID, o qual é utilizado para recuperar a entidade dos índices primários.

Para o armazenamento, o *framework* Object-Injection é organizado em quatro módulos de abstração (CARVALHO, 2013):

- *Módulo de Metaclasses* (Seção 2.3.1): define a interface de implementação para a indexação de entidades e chaves;
- *Módulo de Armazenamento* (Seção 2.3.2): define as estruturas de indexação disponíveis para entidades e chaves;
- *Módulo de Blocos* (Seção 2.3.3): define como os dados são gravados nos meios de armazenamento;
- *Módulo de Dispositivos* (Seção 2.3.4): define os meios computacionais que armazenam as estruturas de indexação.

2.3.1 Módulo de Metaclasses

O Módulo de Metaclasses define a integração da aplicação do usuário com o *framework* Object-Injection através da implementação de interfaces (CARVALHO, 2013). Para realizar a indexação de entidades e chaves, a classe de usuário deve implementar as interfaces ***Entity*** e ***Key*** através de cadeia de heranças. As entidades derivadas de `Entity` devem implementar as operações de serialização, desserialização e comparação através da sobrecarga dos métodos `pushEntity()`, `pullEntity()` e `isEqual()`, respectivamente. As entidades devem ainda possuir dois atributos do tipo `Uuid` (CARVALHO, 2013): o primeiro, `uuid`, identifica o objeto e o segundo, `classId`, é estático e identifica a classe, garantindo que os dados recuperados sejam uma instância da classe correta.

A implementação da interface ***Key*** deve realizar as operações de serialização e desserialização das chaves através da sobrecarga dos métodos `pushKey()` e `pullKey()`, respectivamente. A maneira como as chaves são indexadas ditam quais métodos são sobrecarregados para as operações de comparação. Conforme mostrado pela Figura 4, a interface `Key` é especializada nas interfaces `Order` — que realiza a ordenação das chaves — e `Metric` — que relaciona os dados de acordo com uma medida de proximidade ou similaridade a partir de uma função de distância ou métrica (CARVALHO, 2013). A interface `Metric`, por sua vez, é especializada para a indexação de caracteres, pontos e retângulos pelas interfaces `Edition`, `Point` e `Rectangle`, respectivamente (CARVALHO, 2013).

Um exemplo de utilização do Módulo de Metaclasses é exposto na Figura 4. A classe `City` contém os dados que o usuário deseja persistir. O índice primário é construído a partir da classe `EntityCity`, e são definidos quatro índices secundários. A classe `OrderNameCity`, derivada de `Order`, realiza a indexação secundária (atributo + UUID do objeto) ordenando os dados pelo nome da cidade. A classe `PointLatitudeLongitudeCity` representa a cidade como um ponto, e a sobrecarga do método `distanceTo()` calcula a distância entre duas cidades. A classe `RectangleLatitudeLongitudeCity` representa a cidade como uma região em um espaço e, assim como `PointLatitudeLongitudeCity`, pode ser indexada tanto por uma Árvore M como por uma Árvore R. Por fim, a classe `EditionNameCity` indexa os dados pela similaridade entre os nomes das cidades e é indexada em uma Árvore M.

Também integram este módulo classes auxiliares que disponibilizam métodos para as operações de serialização e desserialização de tipos primitivos, utilizados pelo usuário para persistir e recuperar suas entidades/chaves. O uso exclusivo de tipos primitivos obriga a serialização/desserialização de cada um dos atributos separadamente, garantindo

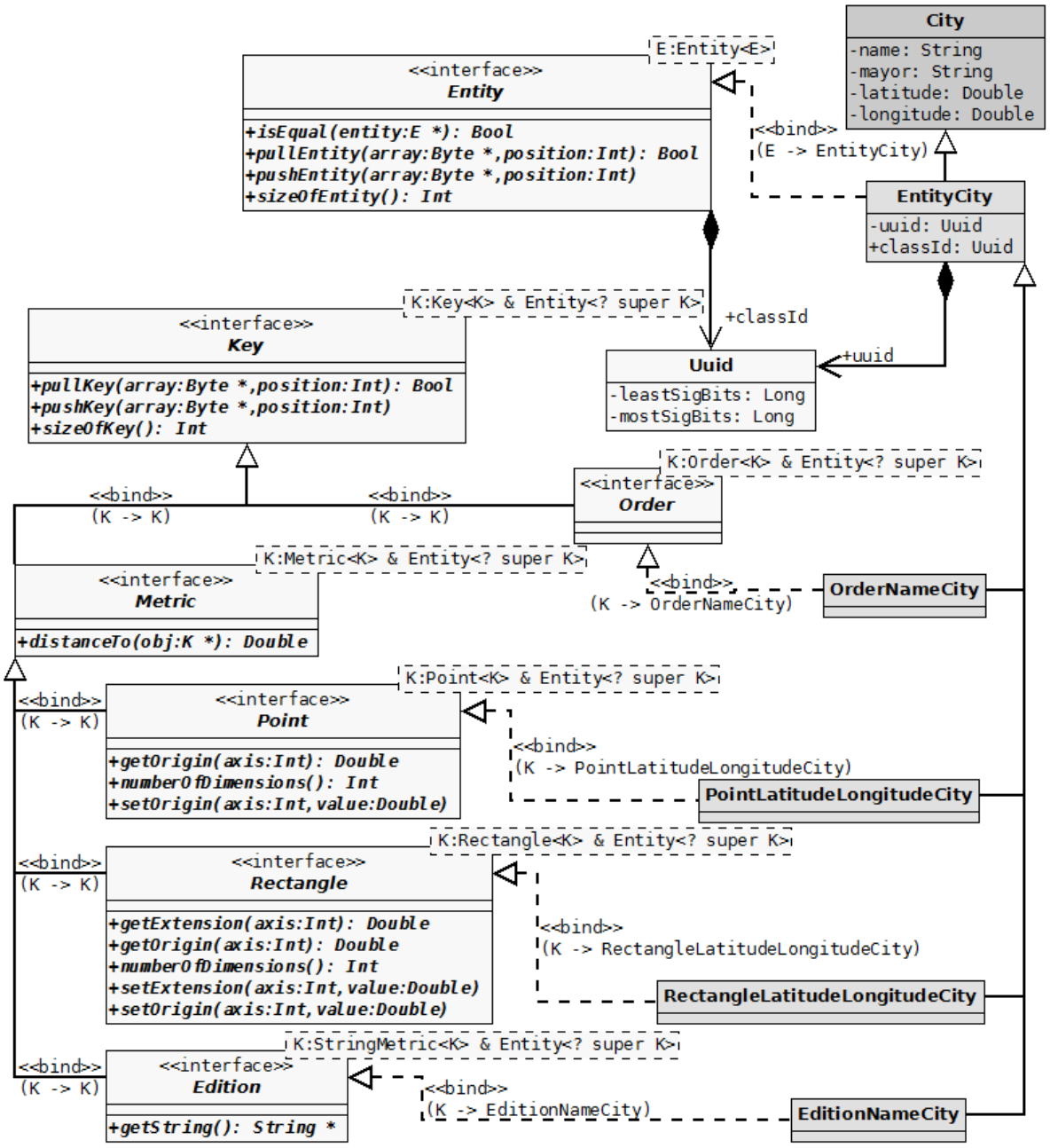


Figura 4: Hierarquia e utilização das interfaces contidas no Módulo de Metaclasses do *framework* Object-Injection (CARVALHO, 2013). Os índices primários são definidos através da interface ***`Entity`*** e os índices secundários são definidos pela interface ***`Key`***.

a integridade dos dados independentemente da arquitetura utilizada (ver Seção 2.4.4). A hierarquia das classes auxiliares é exibida pela Figura 5.

A classe `Stream` encapsula um vetor de *bytes* e a posição inicial (atributo `position`) das operações de serialização ou desserialização realizadas, respectivamente, pelas especializações `PushStream` e `PullStream`. Após cada operação, o atributo `position` é atualizado de acordo com o tamanho do atributo, determinado pelo escopo de `Stream`. São

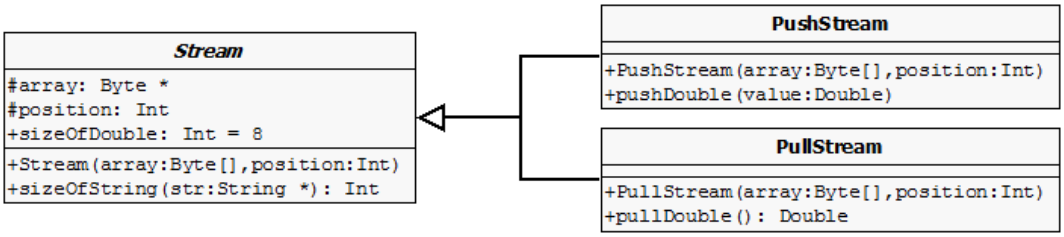


Figura 5: Hierarquia de serialização e desserialização no Módulo de Metaclasses do *framework* Object-Injection (CARVALHO, 2013). Os métodos das classes `PushStream` e `PullStream` são utilizados, respectivamente, para a serialização e desserialização dos atributos das entidades.

disponibilizadas nesta hierarquia métodos para todos os tipos primitivos de dados e para o UUID. Na Figura 5, são destacadas as operações com dados do tipo `Double`.

2.3.2 Módulo de Armazenamento

O Módulo de Armazenamento define as estruturas disponíveis para indexação dos dados do usuário. Sua hierarquia é mostrada na Figura 6, com alguns métodos e atributos omitidos para melhor visualização. A classe `Structure` é a base de todo o módulo, e a partir dela são derivadas três classes: `AbstractStructure`, `EntityStructure` e `KeyStructure`, que definem as operações básicas necessárias para a implementação de estruturas de indexação. A classe `AbstractStructure` é dependente da classe `Workspace` do Módulo de Dispositivos (Seção 2.3.4), garantindo a existência do dispositivo de armazenamento quando a estrutura de indexação é criada. A classe `AbstractEntityStructure` define índices primários que armazenam e recuperam os valores de todos os atributos de uma entidade, enquanto a classe `AbstractKeyStructure` define índices secundários qe armazenam e recuperam somente valores das chaves de busca.

No momento, são disponibilizadas duas estruturas de indexação de entidades — `Sequential`, que realiza o armazenamento de forma sequencial, e `BTreeEntity`, que realiza a indexação em uma Árvore B — e três estruturas de indexação de chaves — `Btree`, `MTree` e `RTree`, as quais indexam as chaves, respectivamente, em uma Árvore B, Árvore M e Árvore R. Todas as estruturas fornecem os métodos necessários para a indexação, persistência e recuperação utilizando os blocos de dados definidos pelo Módulo de Blocos (Seção 2.3.3).

Novas estruturas devem ser derivadas da classe `AbstractEntityStructure` para a indexação de entidades ou da classe `AbstractKeyStructure` para a indexação de chaves.

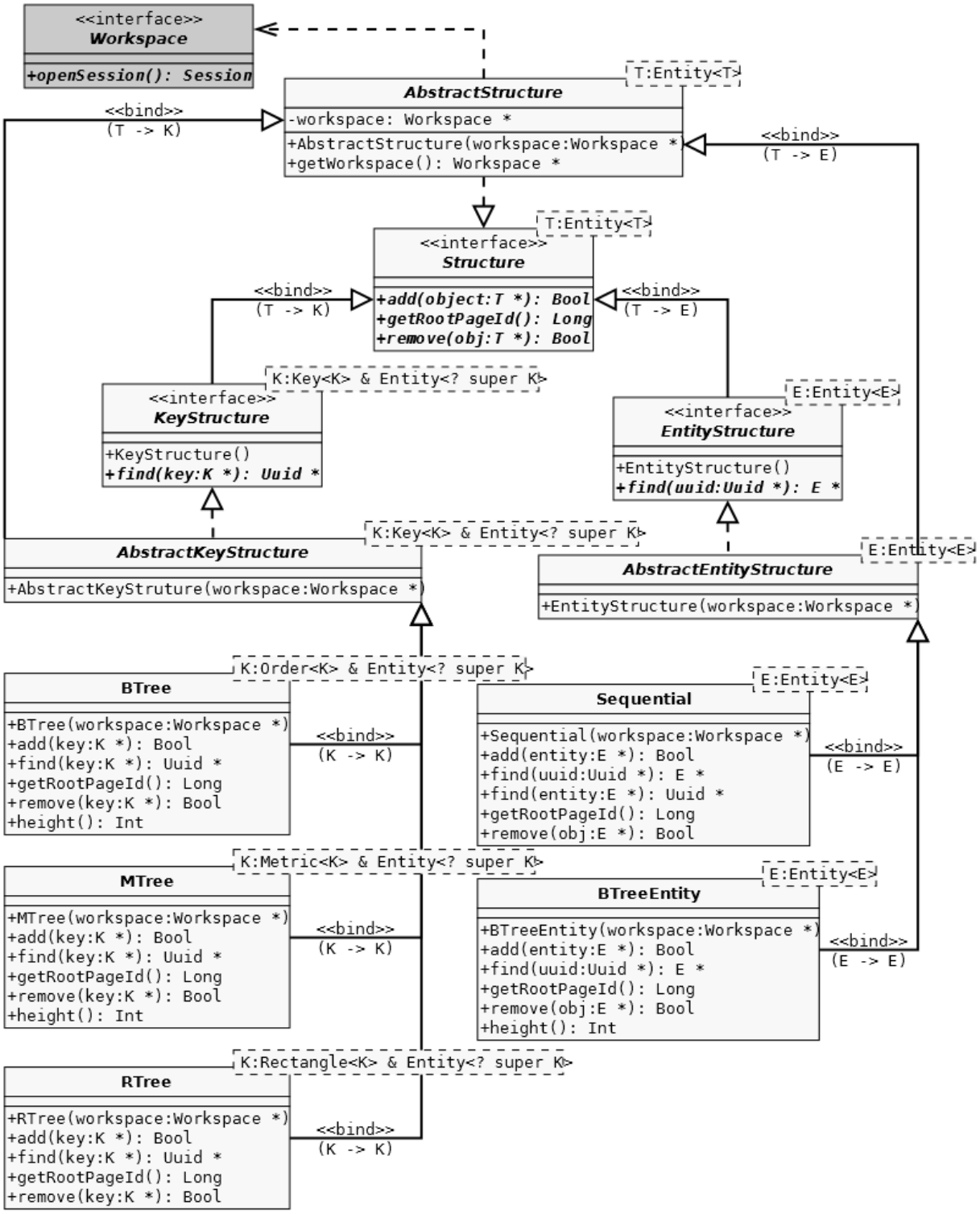


Figura 6: Hierarquia do Módulo de Armazenamento do *framework* Object-Injection (CARVALHO, 2013). As estruturas derivadas de `AbstractEntityStructure` realizam a indexação de entidades, enquanto que as derivadas de `AbstractKeyStructure` realizam a indexação de chaves.

2.3.3 Módulo de Blocos

O Módulo de Blocos define como os dados são organizados em um vetor denominado *bloco*. Um *bloco* é a unidade básica de transferência de dados entre o Object-Injection e o

meio de armazenamento (CARVALHO, 2013).

A classe `Node` (Figura 7) é a classe base da qual todas as classes do módulo são derivadas e implementa as operações básicas para recuperar e armazenar os dados das entidades — na Figura 7, são explicitadas as operações de escrita e leitura somente de dados do tipo `Double`[2]. O atributo `array` é um vetor de *bytes* de tamanho definido pelo atributo `size` que representa o bloco de memória cuja página correspondente é `pageId`.

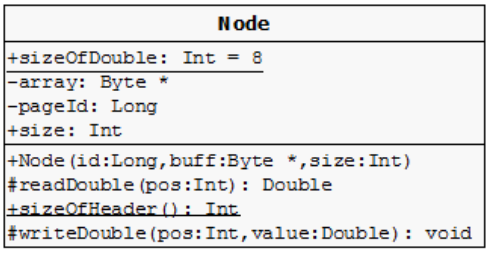


Figura 7: Implementação da classe `Node` no Módulo de Blocos do *framework* Object Injection. O atributo `array` é um vetor de *bytes* de tamanho definido pelo atributo `size` que representa o bloco de memória cuja página correspondente é `pageId`. Nesta classe, também são definidos o tamanho de cada tipo de variável.

A classe `Node` é especializada de forma que cada estrutura de indexação organize seus blocos conforme necessário (CARVALHO, 2013). São criados quatro tipos principais de blocos, conforme mostrado pela Figura 8. A classe `HeaderNode` encapsula o *Bloco Cabeçalho* (Seção 2.3.5), único para cada `workspace` (Seção 2.3.4). A classe `DescriptorNode` é a classe base dos *Blocos Descritores* (Seção 2.3.5), que realizam o armazenamento de metadados das estruturas. Esta classe é especializada para cada estrutura, gerando classes como `SequentialDescriptor` ou `BTreeDescriptor`, omitidas para melhor visualização. Por fim, as classes `EntityNode` e `KeyNode` devem ser especializadas para a criação dos blocos utilizados na indexação de entidades e chaves, respectivamente.

Todos os blocos derivados de `Node` possuem um cabeçalho (*header*) com informações mostrado pela Figura 9. O *Node Type* é um identificador do tipo do bloco e cada classe derivada de `Node` possui um *Node Type* único, garantindo que a estrutura de dados manipule somente os blocos que a compõem (CARVALHO, 2013). Os campos *Previous Page ID* e *Next Page ID* apontam para os blocos contíguos de mesmo nível hierárquico na

[2]O método `readDouble`() recebe como parâmetro a posição inicial do dado em `array` e retorna um valor `Double` lido, enquanto o método `writeDouble`() recebe como parâmetros o valor e em qual posição do `array` ele deve ser escrito. A quantidade de *bytes* utilizada para a representação do valor `Double` é dada pela constante estática `sizeOfDouble`. Estas operações de leitura e escrita são replicadas para todos os tipos primitivos de dados e também para o UUID. Entretanto, não é responsabilidade de `Node` verificar a consistência dos dados salvos ou recuperados (CARVALHO, 2013).

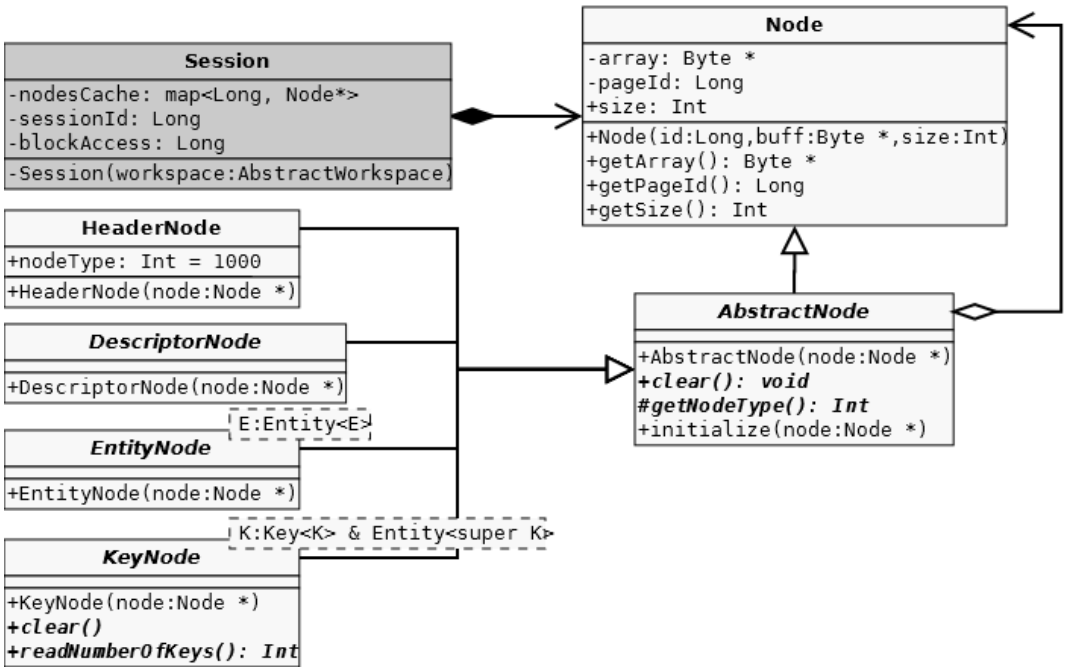


Figura 8: Hierarquia do Módulo de Blocos do *framework* Object-Injection. Existem quatro tipos de blocos principais: (1) `HeaderNode`, que encapsula o Bloco Cabeçalho; (2) `DescriptorNode`, que encapsula os Blocos Descritores; (3) `EntityNode`, utilizado para construção das estruturas de indexação de entidades e `KeyNode`, utilizado para construção das estruturas de indexação de chaves.

estrutura de dados, formando uma lista circular (CARVALHO, 2013). O restante dos *bytes* do bloco são utilizados pelas especializações de `Node` conforme necessário.

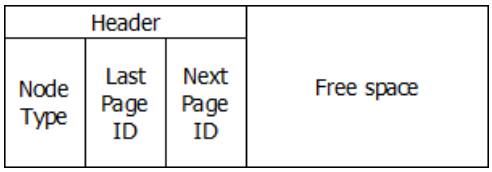


Figura 9: Cabeçalho (*header*) dos blocos do *framework* Object-Injection (CARVALHO, 2013). Os *bytes* livres (`Free space`) são organizados conforme o tipo de bloco especializado.

2.3.4 Módulo de Dispositivos

O Módulo de Dispositivos fornece a interface do *framework* com o meio físico, isolando as estruturas dos meios de armazenamento (CARVALHO, 2013). Sua hierarquia é mostrada pela Figura 10. Algumas funções e atributos foram omitidos para melhor visualização.

A classe `AbstractWorkspace` é a interface que os meios de armazenamento devem especializar para que sejam utilizados pelo Object-Injection (CARVALHO, 2013). Os métodos

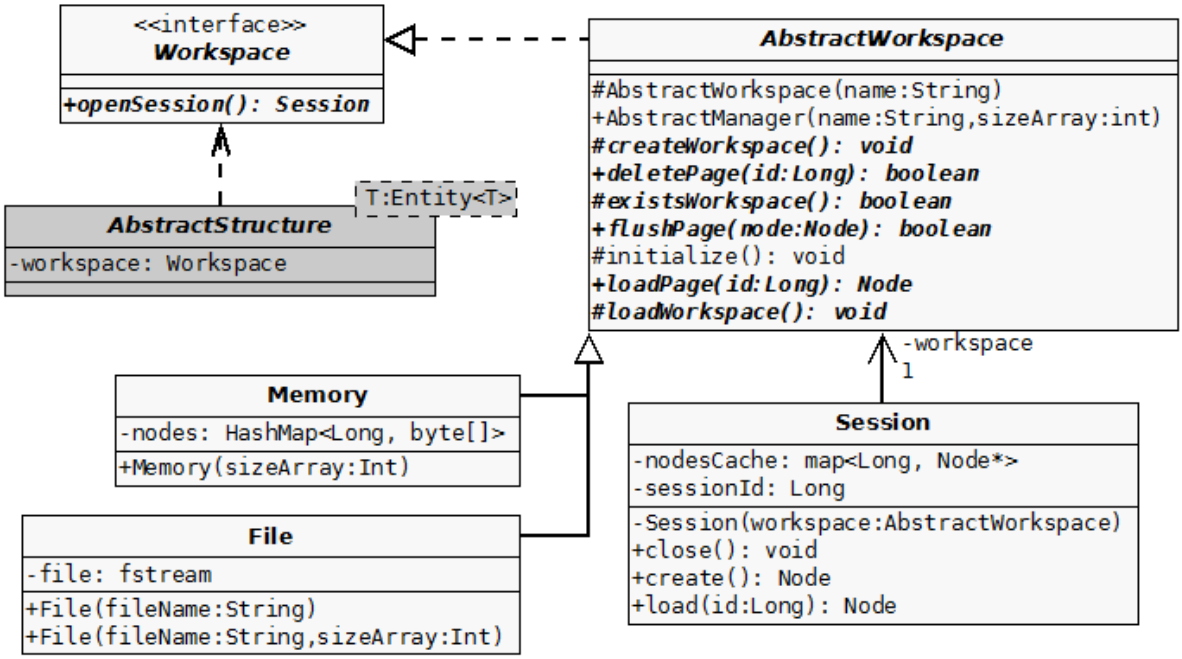


Figura 10: Hierarquia do Módulo de Dispositivos do *framework* Object-Injection. Novos dispositivos de armazenamento devem ser derivados da classe `AbstractWorkspace`.

virtuais `createWorkspace()`, `existsWorkspace()` e `loadWorkspace()` são responsáveis por criar, verificar a existência e carregar o ambiente do meio de armazenamento, inicializando os atributos necessários. As operações de leitura e escrita de dados (implementadas pelos métodos `loadPage()` e `flushPage()`) são específicas para cada tipo de dispositivo e, portanto, devem ser sobrecarregadas. Atualmente, a classe `AbstractWorkspace` é especializada nas classes `File` — a qual armazena os dados de forma persistente, em arquivo — e `Memory` — a qual manipula os dados de forma volátil, por meio de uma estrutura *hash* em memória. Qualquer novo dispositivo de armazenamento incluído no *framework* Object-Injection deve ser derivado da classe `AbstractWorkspace`.

A classe `Session` controla as operações de leitura e escrita de blocos nos dispositivos de armazenamento. Os métodos `create()` e `load()` são utilizados, respectivamente, para adicionar um novo bloco ou carregar um bloco existente. Blocos criados ou carregados na sessão são inseridos em uma estrutura *hash* (`nodesCache`) para diminuir o número de acessos ao dispositivo de armazenamento. O método `close()` finaliza a sessão e atualiza os blocos presentes em `nodesCache` no dispositivo de armazenamento. Toda vez que uma instância de `AbstractWorkspace` é acessada pela classe `Session`, o atributo `sessionId` — um identificador da sessão armazenado no Bloco Cabeçalho (`HeaderNode`) — é atualizado.

Por fim, a classe `Workspace` fornece a interface pela qual o Módulo de Armazenamento (representado na Figura 10 pela classe `AbstractStructure`) pode acessar as operações

relativas aos dispositivos de armazenamento. Esta implementação permite que as estruturas de armazenamento derivadas da classe `AbstractStructure` invoquem os métodos das classes derivadas de `AbstractWorkspace` sem conhecimento de qual meio de armazenamento está sendo utilizado (CARVALHO, 2013).

2.3.5 Organização dos Blocos no Workspace

O Object-Injection segue uma estrutura definida de blocos para armazenamento de metadados e dados indexados. Cada `workspace` criado possui um Bloco Cabeçalho e, para cada estrutura de dados inserida nele, um Bloco Descritor e quaisquer blocos de estrutura necessários para a indexação dos dados. Os Blocos Cabeçalho e Descritores armazenam metadados do `workspace` e das estruturas, enquanto os blocos restantes armazenam a organização das estruturas de persistência de entidades ou de indexação. A Figura 11 exemplifica um `workspace` que utiliza um dispositivo de arquivo em disco contendo duas estruturas de dados distintas. O arquivo é segmentado em blocos de tamanho fixo identificados por um único `PageId`. Novos blocos são concatenados ao `workspace` à medida que mais dados são indexados.

Figura 11: Constituição de um `workspace` com duas estruturas. Cada `workspace` é constituído por um Bloco Cabeçalho único, e, para cada estrutura, um Bloco Descritor e quaisquer outros blocos que a constituam.

Bloco Cabeçalho

O Bloco Cabeçalho ou *Header Node* (Figura 12) é um bloco único no `workspace` que armazena o tamanho de todos os blocos (*Array Size*), o ID do último bloco (*Last Page ID*) do `workspace`, a última sessão que acessou o `workspace` (*Last Session ID*), a quantidade de entradas armazenadas (*Number of Entries*) e a localização dos seus respectivos blocos descritores (seção *Entries*). Cada estrutura de indexação é identificada por um UUID próprio (*Descriptor Node UUID*) e sua localização no `workspace` (*Descriptor Page ID*).

O tamanho dos blocos é definido na criação do `workspace` e não pode ser alterado. Os outros dados são modificados à medida que novos dados ou estruturas são inseridos.

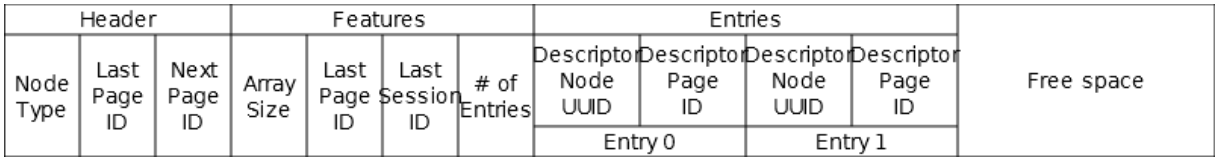


Figura 12: Estrutura do Bloco Cabeçalho de cada `workspace` do *framework* Object-Injection.

Bloco Descritor

O Bloco Descritor ou *Descriptor Node* é único para cada estrutura inserida em um `workspace` e é utilizado para o armazenamento de metadados. Atualmente, os blocos descritores das estruturas de indexação em árvore (`BTreeEntity`, `BTree`, `MTree` e `RTree` — Figura 13) armazenam o ID da raiz da árvore (*Root Page ID*) e a altura da árvore (*Tree Height*). O bloco descritor da estrutura `Sequential` armazena o ID do primeiro e último blocos da estrutura.

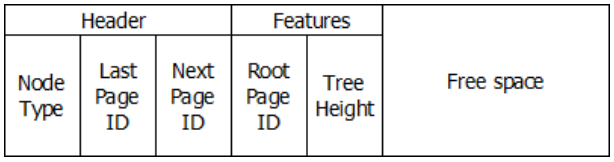


Figura 13: Estrutura dos Blocos Descritores do *framework* Object-Injection para estruturas de indexação em árvore. São armazenadas a altura (*Tree Height*) e a localização da raiz da estrutura (*Root Page ID*). Na estrutura `Sequential`, é armazenada a localização da primeira (*First Page ID*) e última (*Last Page ID*) páginas.

Blocos das Estruturas

Os demais blocos inseridos no `workspace` são dependentes da implementação de cada estrutura de indexação. Para as estruturas de indexação em árvore, existem geralmente dois tipos de blocos: o Bloco Índice ou *Index Node* — o qual armazena os níveis mais altos da árvore — e o Bloco Folha ou *Leaf Node*, que armazena os dados indexados no nível mais baixo da árvore.

2.4 Portabilidade de Software

Portabilidade é "a facilidade com a qual um sistema ou componente pode ser transferido de um ambiente de *hardware* ou *software* para outro" (IEEE Std 610, 1991). Vários

fatores afetam a portabilidade de um *software*, tais como ferramentas e bibliotecas utilizadas no desenvolvimento, compiladores, sistemas operacionais e plataforma de *hardware* (HOOK, 2005; LOGAN, 2007). Na maioria dos casos, os erros de portabilidade surgem quando o programador realiza o desenvolvimento partindo de pressupostos que deixam de ser válidos quando se transfere o *software* para outra plataforma (HOOK, 2005). Este capítulo discute alguns pontos que podem causar problemas devido a diferentes processadores e compiladores. Outros aspectos que podem comprometer a portabilidade do software, como interface de usuário e protocolos de rede, não serão cobertos por não se aplicarem ao presente trabalho.

2.4.1 Alinhamento de Memória

Alguns processadores exigem que os acessos de memória sejam *alinhados*, ou seja, para o processador acessar uma variável de n bytes, o endereço desta variável deve ser um múltiplo de n. O acesso desalinhado pode levar à interrupção do software ou à leitura incorreta de dados (HOOK, 2005), dependendo do processador.

Para não prejudicar a portabilidade, o acesso à memória de forma alinhada deve ser sempre considerado, e operações como conversão de ponteiros devem ser evitadas. Um dos erros de alinhamento de memória mais comuns ocorre quando um *buffer* de memória é acessado através de uma conversão ilegal de ponteiro (HOOK, 2005).

2.4.2 Ordenação de Bytes na Memória

Uma outra diferença existente entre processadores é a ordem na qual os dados de mais de um *byte* são armazenados na memória (*endianess*). A arquitetura é dita *little-endian* quando o *byte* menos significativo é armazenados no endereço de menor valor e *big-endian* quando o *byte* mais significativo é armazenado no endereço de menor valor (NULL; LOBUR, 2003). Assim, para uma variável de quatro *bytes* de valor `0x12FC42A0`, a arquitetura *little-endian* armazena consecutivamente na memória [`0xA0` — `0x42` — `0xFC` — `0x12`], enquanto a arquitetura *big-endian* armazena consecutivamente na memória [`0x12` — `0xFC` — `0x42` — `0xA0`].

A diferença na ordenação dos bytes pode causar problemas quando programas de sistemas com ordenação distintas compartilham dados (NULL; LOBUR, 2003). Para mitigar o problema, é possível padronizar a ordenação dos *bytes* — por exemplo, determinando que os dados compartilhados serão sempre *little-endian* — ou inserir juntamente com a

informação compartilhada um metadado que identifica qual a ordenação utilizada, possibilitando que os diferentes sistemas façam as adaptações necessárias para recuperar a informação corretamente (HOOK, 2005). Se a padronização de escrita em arquivo é feita em *little-endian*, o programa que é executado em arquitetura *little-endian* pode ler os dados diretamente, enquanto o programa executado em arquitetura *big-endian* deve rotacionar os *bytes* lidos de forma a consolidar o valor de maneira correta, e vice-versa.

2.4.3 Tamanho e Implementação de Tipos Primitivos

As normas das linguagens C (ISO/IEC, 2011) e C++ (ISO/IEC, 2014) deixam vários aspectos relativos à definição de tipos primitivos em aberto. Por isso, seu tamanho e comportamento podem variar dependendo da arquitetura do processador e do compilador utilizado (HOOK, 2005), devendo ser considerados no desenvolvimento do *software*. Alguns pontos são discutidos a seguir.

2.4.3.1 Sinalização de Caracteres

Uma variável do tipo `char` deve ser capaz de armazenar qualquer item do conjunto básico dos caracteres de execução, e suas variantes `signed` e `unsigned` têm a mesma capacidade de armazenamento. Entretanto, é dependente da aplicação se uma variável do tipo `char` "simples" (sem os modificadores `signed` e `unsigned`) assume um valor sinalizado ou não. Embora a representação de caracteres não seja afetada, o armazenamento de valores numéricos pode ser comprometido se o programador pressupor que as variáveis assumem valores não sinalizados quando na verdade o sistema realiza o contrário.

2.4.3.2 Codificação de Caracteres

Strings ou cadeias de caracteres são provavelmente os tipos de variáveis mais comuns depois dos tipos inteiros (HYDE, 2006). O padrão ASCII (da norma ISO/IEC 646:1991) se tornou muito limitado para realizar a representação computacional da grande quantidade de caracteres presente em várias línguas (HYDE, 2006), tornando necessária a utilização de novos padrões, como o ISO 8859-1 (Latin 1) e Unicode.

O padrão Unicode — atualmente na versão 7.0 — é compatível reversamente com padrões mais antigos e apresenta conformidade com o padrão ISO/IEC 10646 (*Information technology — Universal Coded Character Set (UCS)*). O Unicode prevê três formas de codificação de caracteres (THE UNICODE CONSORTIUM, 2014):

- **UTF-32:** codifica os caracteres em um número de 32 *bits*. É considerado o padrão mais simples do Unicode, sendo utilizado em muitas plataformas *Unix*. Seu uso é preferível quando o espaço de armazenamento não é uma preocupação.
- **UTF-16:** codifica os caracteres em uma ou duas unidades de 16 *bits*. Descendente histórico das primeiras formas do Unicode, o UTF-16 otimiza a representação dos caracteres pertencentes ao *BMP — Basic Multilingual Plane —*, conjunto que compreende a maioria dos caracteres mais utilizados nos idiomas modernos. Para caracteres do BMP, o padrão UTF-16 funciona como uma codificação de tamanho fixo, utilizando metade da memória do padrão UTF-32.
- **UTF-8:** codifica os caracteres em quantidade variável de unidades de 8 *bits*. O principal objetivo de sua criação foi definir uma codificação que representasse todos os caracteres Unicode sem modificar ou reutilizar nenhum código ASCII, cujas representações permanecem inalteradas na codificação UTF-8.

A Figura 14 mostra a codificação de diferentes caracteres nos padrões estabelecidos pela norma Unicode. Nota-se que só existe compatibilidade entre os padrões UTF-16 e UTF-32 caso o caractere seja representado em apenas uma unidade de 16 *bits*. Da mesma forma, a compatibilidade entre o padrão UTF-8 e os demais só existe para caracteres representados em apenas uma unidade de 8 *bits*, enquanto a relação de conversão para os demais caracteres é mais complexa.

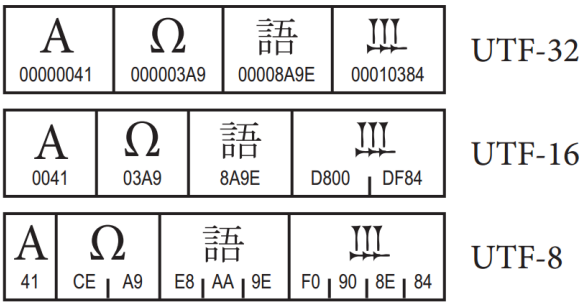


Figura 14: Padrões de codificação de caracteres definidos na norma Unicode (THE UNICODE CONSORTIUM, 2014).

As normas das linguagens C e C++ são vagas em relação à codificação de caracteres. Embora seja especificado que as *strings* com prefixo `u8` (como `u8"abc"`) estejam codificadas utilizando o padrão UTF-8 e sejam representadas pelo tipo `char` (ISO/IEC, 2011), não há como garantir que qualquer *string* composta de uma cadeia do tipo `char` esteja utilizando este esquema de codificação. Na verdade, a maioria das funções da biblioteca `string.h` e seu correspondente `<cstring>` no C++ são dependentes da configuração

da localização (`locale.h`).

Ainda que as versões mais novas das normas prevejam tipos de caracteres de 16 e 32 *bits* — `char16_t` e `char32_t`, respectivamente —, a codificação utilizada para esses tipos — assim como para o tipo `wchar_t`, de tamanho não definido — não é normatizada e, portanto, depende do compilador utilizado.

Assim, cabe ao desenvolvedor garantir a portabilidade de codificação utilizando por exemplo as propriedades de localização. Infelizmente, os nomes utilizados nestas funções não são normatizados e podem variar conforme a plataforma, impedindo que ajustes sejam feitos de forma automática.

2.4.3.3 Sinalização de Tipos Inteiros

As normas das linguagens especificam que os tipos inteiros sinalizados devem possuir um *bit* de sinalização. Quando este *bit* é zero (número positivo), o valor do número não é alterado. Entretanto, o número negativo pode ser representado de três maneiras: *sinal e magnitude* — o *bit* de sinalização é setado, mas o módulo do número continua o mesmo —, *complemento para um* — o número é negado *bit* a *bit* — e *complemento para dois* — o número é negado *bit* a *bit* e se adiciona um ao resultado. Tal liberdade de implementação pode causar erros se números negativos forem compartilhados entre plataformas com padrões de sinalização distintos.

2.4.3.4 Tamanho de Tipos Inteiros

Uma variável do tipo `int` possui o chamado *tamanho natural* da arquitetura, correspondente ao tamanho dos registradores internos. Tal correspondência tem como objetivo a obtenção de performance ótima nas operações (HOOK, 2005). Embora as variáveis inteiras "simples" possuam tamanhos conhecidos uma vez que se saiba o processador utilizado, é dependente do compilador o tamanho das variantes `short`, `long` e `long long` (LOGAN, 2007). O compilador deve ajustar os limites mínimo e máximo das variáveis inteiras, suas variantes e suas correspondências não sinalizadas em arquivo de biblioteca correspondente (`limits.h` na linguagem C e os *templates* `numeric_limits` e arquivo `<climits>` na linguagem C++).

O tamanho não definido de variáveis também pode causar problemas na criação de constantes e máscaras. Uma constante de valor `0xFFFFFFFF` possui valor -1 em um sistema com inteiros de 32 *bits* sinalizados em complemento para dois. Se portada para um sistema

de 64 *bits*, passa a ter um valor positivo (HOOK, 2005).

Caso seja necessário garantir o tamanho de uma variável, tipos com tamanhos pré-definidos devem ser utilizados (HOOK, 2005). As normas das linguagens especificam tipos inteiros de tamanho padronizado, disponibilizados nos arquivos `stdint.h` na linguagem C e seu correspondente em C++ `<cstdint>`. Nestes arquivos, são especificados três tipos inteiros: *inteiros de tamanho fixo*, *inteiros com tamanhos mínimos* e *inteiros rápidos com tamanhos mínimos*, em conjunto com os limites mínimo e máximo de cada um.

Inteiros de tamanho fixo são definidos a partir dos tipos `intN_t` e `uintN_t`, representando, respectivamente, inteiros sinalizados e não sinalizados com exatamente `N` *bits* de tamanho. A definição dos tipos de tamanho fixo é opcional.

Inteiros de tamanho mínimo são definidos como `int_leastN_t` e `uint_leastN_t`, representando inteiros sinalizados e não sinalizados que possuam *pelo menos* `N` *bits*. A definição dos tipos é obrigatória para variáveis de 8, 16, 32 e 64 *bits*.

Inteiros rápidos de tamanho mínimo são definidos a partir dos tipos `int_fastN_t` e `uint_fastN_t`, representando, respectivamente, inteiros sinalizados e não sinalizados que sejam os mais rápidos para realizar operações e tenham no mínimo `N` *bits*. Também neste caso, a definição é obrigatória para variáveis de 8, 16, 32 e 64 *bits*.

2.4.3.5 Representação e Operações de Ponto Flutuante

Os números de ponto flutuante são utilizados para a representação de números reais e são divididos nas partes *exponencial* e *não exponencial*, esta também chamada de *mantissa* (PRATA, 2013). As normas das linguagens especificam três tipos de dados de ponto flutuante: `float`, `double` e `long double`, cujas propriedades de representação e respectivos limites são disponibilizados no arquivo `float.h` na linguagem C, seu correspondente `<cfloat>` e os *templates* `numeric_limits` na linguagem C++. Os números devem ser capazes de armazenar pelo menos seis dígitos significativos e permitir componentes exponenciais de 10^{-37} até 10^{+37} (PRATA, 2013). Entretanto, a representação destes números é definida pela implementação, e fatores como o *hardware*, compilador e até mesmo a maneira como o código é construído afetam os resultados obtidos com essas variáveis (LOGAN, 2007). Assim, o desenvolvedor deve estar atento aos possíveis erros que possam ocorrer devido à incerteza na precisão e consistência de variáveis deste tipo, especialmente quando efetuando comparações (HOOK, 2005).

Foi desenvolvido pelo IEEE a norma 754 (IEEE Std 754, 2008), que padroniza a repre-

sentação de tipos de ponto flutuante e suas operações aritméticas. Em 2011 a norma foi adotada internacionalmente como ISO/IEC/IEEE 60559:2011 (PRATA, 2013). As normas das linguagens C e C++ especificam que o suporte à norma 754 é opcional. Entretanto, caso ele exista, o tipo `float` corresponde ao formato de precisão simples da norma, enquanto o tipo `double` corresponde ao formato de precisão dupla da norma.

No caso específico de compartilhamento de dados de ponto flutuante, o primeiro desafio é extrair os dados da variável, o que poderia ser feito por meio de uma estrutura `union` ou, mais asseguradamente, por meio de um *casting* para `(void *)` ou `(unsigned char *)` (HOOK, 2005). Ainda assim, a maneira como os *bits* representam o número pode variar, uma vez que a própria norma 754 prevê cinco maneiras diferentes de representar uma grandeza de ponto flutuante. Mesmo que seja assegurado que a representação dos tipos de ponto flutuante seja a mesma nos diferentes sistemas, é necessário uma grande bateria de testes ou, alternativamente, armazenar os valores em um formato diferente dos tipos de ponto flutuante — por exemplo, no formato ASCII (LOGAN, 2007) — para garantir a portabilidade deste tipo de dado.

2.4.4 Utilização de Dados Compostos

As normas das linguagens C e C++ não normatizam a maneira como as estruturas `struct` e `class` são implementadas na memória. Assim, dependendo dos requisitos de alinhamento da plataforma, *bits de preenchimento* podem ser utilizados, alterando o tamanho que a estrutura ou classe ocupa na memória (HOOK, 2005). Tais diferenças, em conjunto com a incerteza do tamanho de variáveis e sua sinalização e possível incompatibilidade de ordenação de *bytes*, podem causar grande inconveniente ao programador.

As operações de serialização (salvar dados) e desserialização (recuperar dados) são uma fonte comum de problemas no desenvolvimento de sistemas multiplataforma quando realizadas de forma direta, utilizando apenas seu endereço e uma função como `memcpy()` (HOOK, 2005). Para mitigar o problema, o compartilhamento de informações serializadas entre plataformas deve ser de alguma forma padronizado no projeto. As informações sobre ordenação de *bytes*, tamanhos de variáveis e sua sinalização podem ou ser pré-acordadas entre as partes ou embutidas nos dados, permitindo que cada sistema realize as adaptações necessárias para o armazenamento e recuperação corretos (LOGAN, 2007). Além disso, os dados dessas estruturas devem ser serializados um a um, utilizando tipos de variáveis de tamanhos conhecidos, podendo então ser encaminhados para o pretendido meio de transmissão (por exemplo, um arquivo ou um *buffer* de rede) (HOOK, 2005).

2.4.5 Disponibilidade de Recursos Computacionais

A diferença de recursos computacionais entre múltiplas plataformas pode trazer grandes incômodos. Quando é feita a migração de um programa para uma plataforma mais simples ou com recursos mais limitados, o desenvolvedor deve estar atento à disponibilidade e custo computacional das operações — como, por exemplo, operações com dados do tipo `float` em processadores mais simples ou limitações de memória caso grandes estruturas ou *arrays* sejam utilizados (HOOK, 2005).

O uso indiscriminado de variáveis locais ou a utilização de recursão podem trazer problemas relacionados à pilha de *Stack*[3] em plataformas com menos recursos. Tais situações são de difícil análise, uma vez que o erro pode ter ocorrido muito antes de ser sinalizado (HOOK, 2005). Para auxiliar em tais situações, o sistema pode ser desenvolvido com um supervisor de *Stack*, que monitora quanto da pilha foi utilizado em relação a um limite estabelecido. Uma outra técnica para análise de estouro (*overflow*) da pilha é preenchê-la com um valor conhecido, o que torna possível verificar a quantidade de memória utilizada durante a depuração (GANSSLE, 1999).

O desenvolvedor, sabendo dos problemas que a pilha de *Stack* causa, pode então se decidir pelo uso da alocação dinâmica de memória, esta também fonte de problemas (GANSSLE, 1999). O uso de memória dinâmica sem um dispositivo de supervisão de blocos liberados (*garbage collector*) pode levar à fragmentação do *heap* e possível falha do método `malloc()`. Para prevenir maiores danos caso isto ocorra, o programa deve sempre verificar o ponteiro retornado de forma a assegurar que a operação de alocação foi bem-sucedida (GANSSLE, 1999).

2.5 Considerações Finais

Este Capítulo apresentou conceitos relacionados ao desenvolvimento deste trabalho. É feita uma pequena introdução sobre Sistemas Embarcados e o sistema FreeRTOS™ para reforçar os pontos desafiadores no desenvolvimento de aplicações de sistemas com várias restrições, que são consideradas, em seguida, quando se discute os requisitos desejados para uma aplicação gerenciadora de dados construída para operar em sistemas móveis ou embarcados.

[3] A pilha de *Stack* é um buffer LIFO (*Last In, First Out*) utilizada para o armazenamento de variáveis locais e controle do fluxo de programa na chamada de sub-rotinas (BALL, 2002). Tipicamente, o tamanho da pilha de *Stack* é determinado estaticamente no processo de construção do *software* (HOOK, 2005).

São apresentadas em seguida uma descrição em alto nível do *framework* Object-Injection e uma pequena descrição de seus módulos para situar o leitor sobre as interfaces utilizadas nos testes e também sobre as modificações realizadas durante o desenvolvimento.

Por fim, são apresentadas algumas questões de portabilidade de *software* relevantes quando é feita a compatibilidade de linguagens de programação e se busca a portabilidade entre diferentes arquiteturas.

Para a compreensão deste texto, é desejável que o leitor já possua certa familiaridade com a área de estruturas de dados, uma vez que os conceitos de persistência e indexação não são aprofundados. Caso necessário, é possível consultar Carvalho (2013) ou as estruturas de indexação disponíveis no *framework* Object-Injection (Árvores B, M e R) na literatura de estrutura de dados.

3 Framework Object-Injection para Sistemas Embarcados

Este capítulo descreve as atividades realizadas por este trabalho visando a portabilidade do *framework* Object-Injection para o sistema FreeRTOS™.

Primeiramente, foi realizada a implementação na linguagem C++ da estrutura de indexação de entidades `BTreeEntity` baseando-se na implementação da linguagem Java do *framework*. Em seguida, foram analisados aspectos da compatibilidade entre as duas implementações Java e C++ para que o compartilhamento de informação entre elas ocorresse de forma transparente. As implementações realizadas para tal objetivo também contribuiram para melhorar a portabilidade da implementação C++. Em seguida, foram investigados fatores que pudessem melhorar o desempenho do *framework*, como por exemplo a frequência de operações de atualização em disco. Por fim, foi realizada a integração do *framework* Object-Injection com o sistema FreeRTOS™ através da adaptação do código do *framework* para utilização do sistema de arquivos, semáforos e gerenciador de memória dinâmica do sistema FreeRTOS™.

3.1 Armazenamento de Objetos Usando Entidades

Para suportar o armazenamento de objetos foi implementada na linguagem C++ uma estrutura baseada em uma Árvore B chamada `BTreeEntity`, que armazena todos os atributos do objeto e os indexa através de um UUID. A codificação, baseada na sua correspondente em Java, abrangeu a criação da classe `BTreeEntity` no Módulo de Armazenamento (Seção 2.3.2) e das classes `BTreeEntityDescriptor`, `BTreeEntityNode`, `BTreeEntityIndex` e `BTreeEntityLeaf` no Módulo de Blocos (Seção 2.3.3). A relação das classes implementadas com os módulos é mostrada pela Figura 15.

Após a implementação, foram realizados testes funcionais para validar o funcionamento da estrutura. Os testes foram realizados utilizando a classe `Student`, a qual

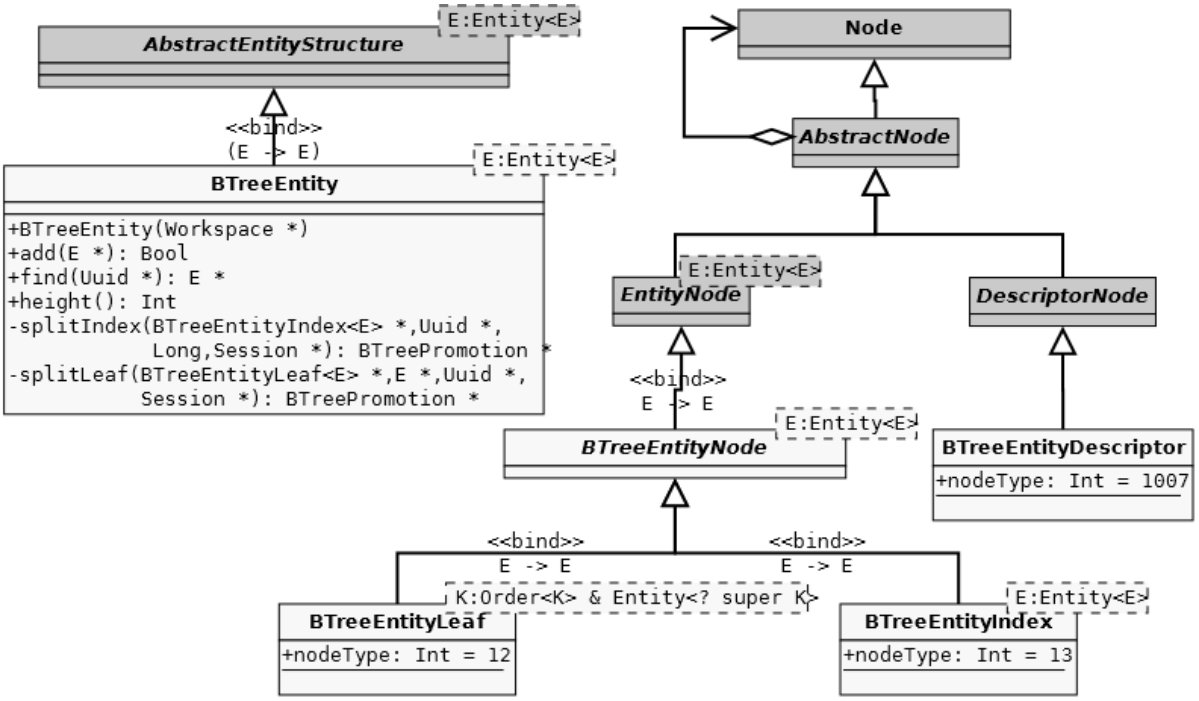


Figura 15: Modelagem da estrutura `BTreeEntity` na arquitetura do *framework* Object-Injection. A classe `BTreeEntity` é inserida no Módulo de Armazenamento (Seção 2.3.2) e as classes dos blocos que a compõem (`BTreeEntityDescriptor`, `BTreeEntityNode`, `BTreeEntityIndex` e `BTreeEntityLeaf`) são inseridas no Módulo de Blocos (Seção 2.3.3).

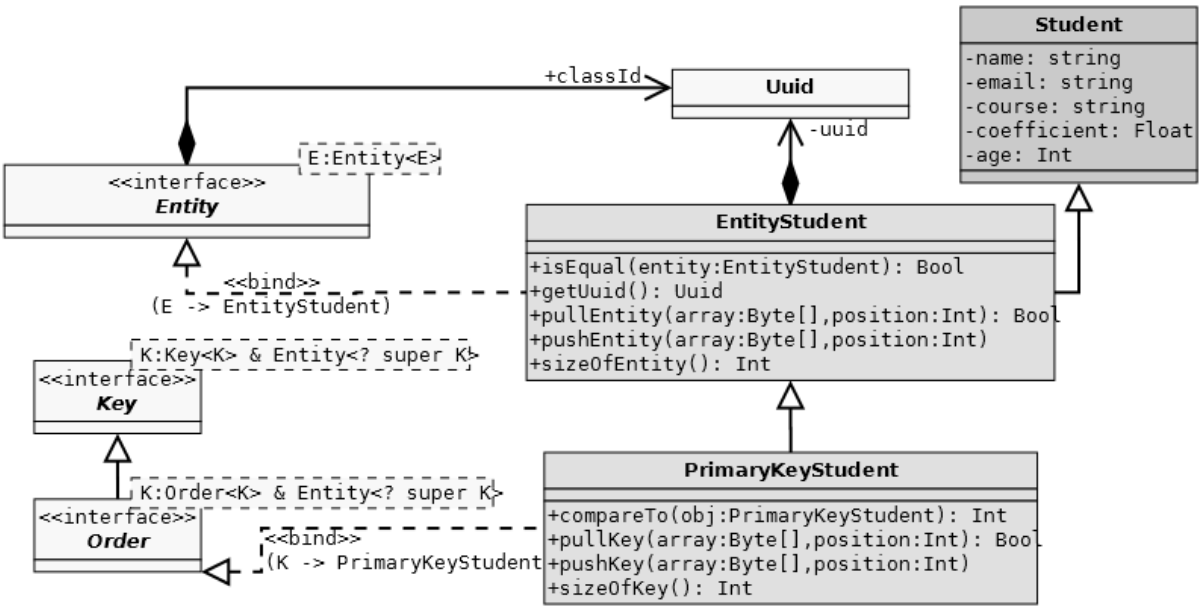


Figura 16: Implementação das interfaces ***`Entity`*** e ***`Key`*** para a classe `Student`.

possui atributos dos tipos `int`, `float` e `string`. Seguindo a implementação das interfaces definidas no Módulo de Metaclasses (Seção 2.3.1), foram criadas as classes derivadas `EntityStudent` para a implementação da interface ***`Entity`*** e `PrimaryKeyStudent` para a

implementação da interface *Key*, indexando as entidades pelo nome do aluno. A Figura 16 exibe as relações entre as classes criadas e o Módulo de Metaclasses.

Um dos problemas enfrentados foi a correção do erro *BUS ERROR* nas operações de leitura e/ou escrita dos dados tipo `Float`. Após análise, verificou-se que a leitura e escrita de dados de ponto flutuante (`Float` e `Double`) nas classes `Node` (Seção 2.3.3), `PushStream` e `PullStream` (Seção 2.3.1) foram feitas através da conversão de um ponteiro em um buffer de *bytes*, conforme exemplificado pelos métodos `readFloat()` e `writeFloat()` da classe `Node` no Programa 1. Tal implementação pode causar a leitura de dados de forma desalinhada (Seção 2.4.1), comprometendo a integridade dos dados ou mesmo interrompendo o programa.

Programa 1: Rotinas originais de leitura e escrita de dados tipo `Float`.

```
Float readFloat(Int pos) {
    Float * buff = (Float *) (array + pos);
    return buff[0];
}
void writeFloat(Int pos, Float value) {
    Float * buff = (Float *) (array + pos);
    buff[0] = value;
}
```

Assim, foi necessário modificar os métodos de serialização e desserialização desses dados de forma a garantir o alinhamento de memória e suprimir o erro encontrado. As novas implementações são apresentadas pelo Programa 2.

Programa 2: Rotinas modificadas de leitura e escrita de dados tipo `Float`.

```
Float readFloat(Int pos) {
        Float value;
        Byte * ptr = (Byte *) & value;
        ptr[0] = (Byte) array[pos + Node::sizeOfFloat - 1];
        ptr[1] = (Byte) array[pos + Node::sizeOfFloat - 2];
        ptr[2] = (Byte) array[pos + Node::sizeOfFloat - 3];
        ptr[3] = (Byte) array[pos + Node::sizeOfFloat - 4];
        return value;
}
void writeFloat(Int pos, Float value) {
        Byte * ptr = (Byte *) & value;
        array[pos + Node::sizeOfFloat - 1] = (Byte) ptr[0];
        array[pos + Node::sizeOfFloat - 2] = (Byte) ptr[1];
        array[pos + Node::sizeOfFloat - 3] = (Byte) ptr[2];
        array[pos + Node::sizeOfFloat - 4] = (Byte) ptr[3];
}
```

Também nesta etapa, foram realizados testes para avaliar a alocação e liberação de memória *heap* da estrutura `BTreeEntity` utilizando a ferramenta Valgrind (VALGRIND,

2014). Todos os vazamentos de memória (*leaks* — alocações não desalocadas posteriormente) foram resolvidos.

3.2 Formalização dos Requisitos do Sistema

Como todo *software*, o *framework* Object-Injection foi implementado a partir de alguns pressupostos a respeito da arquitetura e recursos computacionais disponíveis. Não havia, entretanto, nenhuma análise que impedisse o uso do *software* caso algum de seus pré-requisitos não fosse atendido. Assim, foi criado um arquivo (`config.h`) que realiza tais verificações, gerando um erro de compilação caso alguma condição falhe. A princípio, foram inseridas duas verificações, conforme ilustrado pelo Programa 3. São verificados (1) o número de *bits* que compõem um *byte* (dado utilizado para operações de rotação de variáveis) e (2) a conformidade do compilador com o *standard* C++11, necessária devido ao uso de diretivas `static_assert()`, `is_convertible()` e `is_base_of()`, incluídas apenas nesta versão do *standard*. Para tanto, são testadas as macros `CHAR_BIT` e `__cplusplus`.

Programa 3: Verificações do arquivo `config.h` para garantir a compilação do *framework* Object-Injection.

```
#if ((!defined __cplusplus) || (__cplusplus < 201100)) /* GCC compiler. */
#error "ObI_001: C++ 11 compiler required. Please set -std=c++11 option."
#endif
#if (CHAR_BIT != 8)
#error "ObI_002: Number of bits in a byte must be 8."
#endif
```

3.3 Efetivação da Compatibilidade Entre Implementações

O *IEEE Standard Computer Dictionary* (IEEE Std 610, 1991) define Compatibilidade como a "capacidade de dois ou mais sistemas ou componentes de trocarem informações". No *framework* Object-Injection, a compatibilidade entre implementações permite o compartilhamento de dados de forma transparente, ou seja, quando uma mesma entidade é implementada em sistemas distintos, o arquivo de persistência pode ser utilizado em qualquer implementação, independentemente da linguagem de programação utilizada, para a recuperação das instâncias persistidas. Para tal fim, dois pontos devem ser garantidos: (1) os dados compartilhados devem ser iguais — os tipos de variáveis devem ser os mesmos — e (2) os dados compartilhados devem seguir uma estrutura conhecida — as implementações devem construir arquivos de indexação iguais para o mesmo conjunto de dados.

3.3.1 Padronização do Tamanho e Sinalização de Tipos Primitivos

Uma vez que a linguagem Java define o tamanho e faixa de valores de seus tipos primitivos (GOSLING et al., 2014), a compatibilidade pode ser assegurada caso as variáveis da implementação C++ sejam padronizadas de forma a coincidir com as especificações da linguagem Java, dadas pela Tabela 2. É possível notar que os tipos inteiros possuem tamanho determinado e são sempre sinalizados e que o tipo `char` difere especialmente do utilizado em C++, uma vez que ele possui dois *bytes* de tamanho e segue a codificação UTF-16 (a linguagem Java, entretanto, fornece métodos para codificação de *strings* em outros conjuntos de caracteres — como UTF-8, ASCII e Latin-1).

Tipos Inteiros

Tipo	Tamanho (*bytes*)	Intervalo
`byte`	1	-128 a 127
`short`	2	-32768 a 32767
`int`	4	-2147483648 a 2147483647
`long`	8	-9223372036854775808 a 9223372036854775807
`char`[1]	2	0x0000 a 0xFFFF

[1] O tipo **char** em Java segue a codificação UTF-16.

Tipos de Ponto Flutuante[2]

Tipo	Tamanho (*bytes*)	Expoente (*bits*)	Precisão (*bits*)
`float`	4	8	24
`long`	8	11	53

[2] Os tipos **float** e **double** são associados, respectivamente, aos formatos de precisão simples de 32 *bits* e precisão dupla de 64 *bits* da norma IEEE 754.

Tipo Lógico

Tipo	Tamanho (*bytes*)	Valores
`boolean`	1	`true` ou `false`

Tabela 2: Especificações dos tipos primitivos da linguagem Java (GOSLING et al., 2014).

Para assegurar a compatibilidade entre as variáveis, foram utilizados os tipos customizados do arquivo `types.h` já existentes na implementação C++ do Object-Injection.

A. Tamanho e Sinalização de Tipos Inteiros

Os tipos inteiros customizados `Byte`, `Short`, `Int` e `Long` foram redefinidos utilizando inteiros sinalizados de tamanho fixo para equipará-los, respectivamente, aos tipos `byte`, `short`, `int` e `long` da linguagem Java — tipos inteiros não sinalizados não são utilizados pelo *framework*. No arquivo `types.h` é realizada uma verificação da existência de tipos

inteiros de tamanho fixo testando a definição de seus limites. Caso os tipos não existam, é gerado um erro de compilação, cabendo ao usuário fornecer as diretivas `typedef` e `#define` que definam, respectivamente, as variáveis de tamanho fixo e seus limites mínimo e máximo. O Programa 4 ilustra o código da definição do tipo customizado `Byte`, um inteiro sinalizado de um *byte*.

Programa 4: Definição do tipo inteiro customizado `Byte`.

```
#if (!defined(INT8_MIN) || !defined(INT8_MAX))
#error "ObI_004: Necessary type definition missing (int8_t)."
#else
    typedef int8_t Byte;        /*! @brief Equivalent of byte Java type. */
#endif
```

Como o esquema de sinalização de tipos inteiros é dependente da arquitetura (Seção 2.4.3.3), é necessário verificar o padrão de sinalização utilizado para garantir o correto compartilhamento dos dados. Para isso, uma variável com um valor negativo conhecido é testada por uma declaração `static_assert()`[1] de forma a verificar se a sua representação corresponde ao utilizado pela linguagem Java — complemento para dois. O Programa 5 mostra a declaração feita para testar a sinalização de variáveis. É utilizada uma variável de 8 *bits* com valor -1 e aplicada uma máscara para verificar se a representação do número é feita em complemento de dois.

Programa 5: Verificação do tipo de sinalização de inteiros.

```
const Byte test2Complement = -1;
static_assert((test2Complement & 0xFF) == 0xFF, "ObI_005: Negative
  representation not compatible. Negative values must be represented by
  two-complement.");
```

B. Tamanho e Codificação de Caracteres

Conforme discutido na Seção 2.4.3.2, não é possível garantir de forma automática a codificação de caracteres em diferente sistemas, cabendo ao usuário buscar a melhor solução para a sua aplicação. No escopo deste trabalho, as variáveis do tipo customizado `Char` são criadas a partir do tipo primitivo `char` e têm tamanho de um *byte*. Os testes foram executados utilizando caracteres do conjunto ASCII, cuja representação é a mesma para padrões de codificação populares como o UTF-8 e Latin-1.

[1] Uma declaração `static_assert()` permite a verificação de valores constantes em tempo de compilação. Se a expressão analisada é avaliada como falsa, um erro de compilação é gerado, podendo o programador oferecer uma mensagem de erro ao usuário (DEITEL; DEITEL, 2011).

C. Representação de Tipos de Ponto Flutuante

Os tipos de ponto flutuante customizados `Float` e `Double` foram criados a partir dos tipos `float` e `double` da linguagem C++. Para verificar a conformidade dos tipos nativos com a norma IEEE 754, foi inserida uma declaração `static_assert()` para verificar como é feita a representação dos tipos de ponto flutuante do sistema, conforme mostrado pelo Programa 6.

Programa 6: Teste de representação dos tipos de ponto flutuante.

```
static_assert(((std::numeric_limits<float>::radix == 2) &&
  (std::numeric_limits<float>::digits == 24) &&
  (std::numeric_limits<float>::is_iec559)), "ObI_006: Floating-piont
  representation not compatible. Float and Double types do not follow IEEE
  754 standard.");
```

3.3.2 Padronização da Estrutura de Arquivos

Para verificar a consistência dos arquivos gerados pelas implementações do *framework* Object-Injection, foi criado um arquivo texto com 100 mil instâncias de `Student` (Figura 16), as quais foram indexadas e persistidas, juntamente com suas chaves primárias, em arquivos de índices primários e secundários pela implementação C++. Entretanto, a tentativa de recuperação das mesmas entidades e chaves pela implementação Java não foi bem sucedida, evidenciando uma incompatibilidade entre as implementações. Para localizar sua origem, foi realizada a indexação e persistência das entidades e chaves dos mesmos objetos na implementação Java, de forma a identificar diferenças entre os arquivos de persistência.

Como o *framework* gera arquivos binários de difícil análise, foi desenvolvido na linguagem LabVIEW® o *ObInject File Dumper*, um *software* que lê as informações do arquivo binário e converte o valor de cada *byte* em texto, escrevendo cada uma de suas páginas em um arquivo. O *software*, cuja tela é mostrada pela Figura 17, também permite visualizar o conteúdo de cada um dos arquivos gerados.

A partir da análise dos arquivos texto gerados, foi possível identificar inconsistência entre as operações de serialização/desserialização realizadas pelas classes `Node` (Módulo de Blocos — Seção 2.3.3) e as derivadas da classe `Stream` (Módulo de Metaclasses — Seção 2.3.1), que acessam e modificam o vetor de *bytes* que representa cada bloco no programa.

As classes derivadas de `Stream` realizavam as operações de dados inteiros de mais de

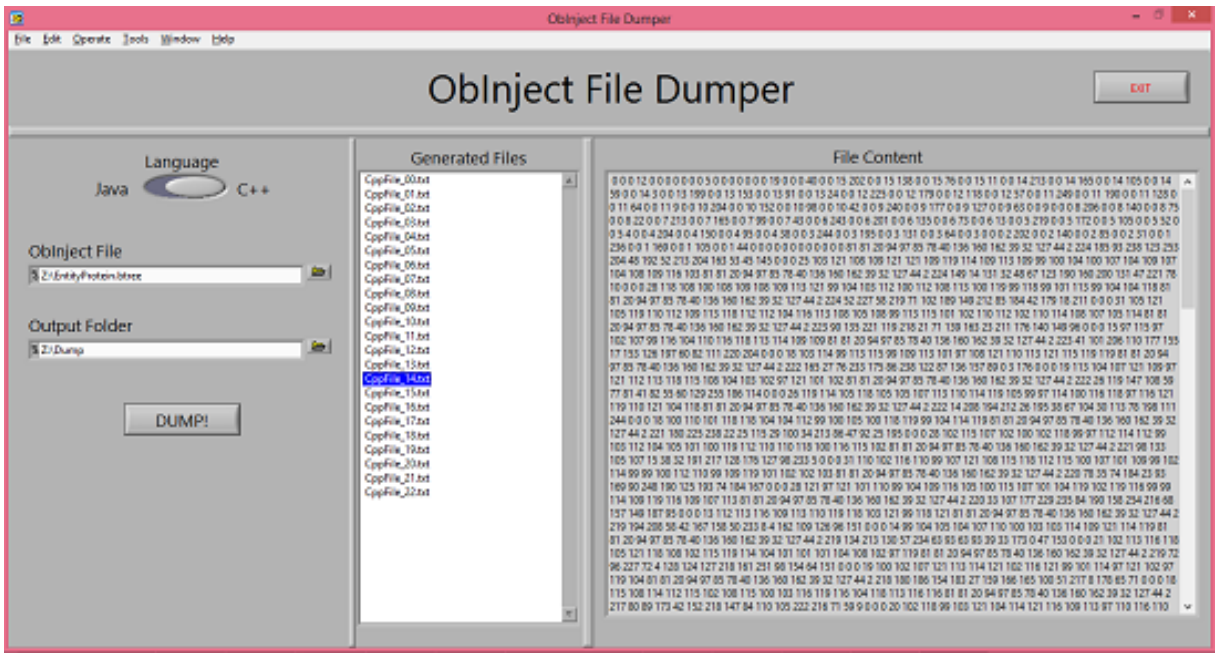


Figura 17: Tela principal do *ObInject File Dumper*.

um *byte* (`Short`, `Int` e `Long`) sempre em ordenação contrária à da arquitetura (conforme ilustrado pelo Programa 7 para o tipo `Int`), enquanto a classe `Node` o fazia por meio de conversão de ponteiros (conforme ilustrado pelo Programa 8, também para o tipo `Int`), seguindo a ordenação dos *bytes* da arquitetura. No teste executado, em arquitetura *Little-Endian*, as derivadas da classe `Stream` armazenaram os dados em *Big-Endian*, enquanto a classe `Node` armazenava os dados em *Little-Endian*.

Programa 7: Serialização do tipo `Int` pela classe `PushStream`.

```
void PushStream::pushInt(Int value) {
    array[position + sizeOfInt - 1] = (Byte) (value & mask);
    value >>= bitsPerByte;
    array[position + sizeOfInt - 2] = (Byte) (value & mask);
    value >>= bitsPerByte;
    array[position + sizeOfInt - 3] = (Byte) (value & mask);
    value >>= bitsPerByte;
    array[position + sizeOfInt - 4] = (Byte) (value & mask);
    value >>= bitsPerByte;
    position += sizeOfInt;
}
```

Tal incompatibilidade traz à tona três questões: (1) operações de serialização e desserialização em classes diferentes são potenciais fontes de erros e inconsistências, (2) a ordenação dos *bytes* dos dados compartilhados deve ser padronizada para garantir a compatibilidade e (3) as implementações devem se adaptar para garantir o armazenamento e recuperação corretos independentemente da arquitetura sob a qual são executadas.

Programa 8: Serialização do tipo `Int` pela classe `Node`.

```
void writeInt(Int pos, Int value) {
    Int * buff = (Int *) (array + pos);
    buff[0] = value;
}
```

A. Centralização das Operações de Serialização e Desserialização

Para centralizar as operações de serialização e desserialização dos dados primitivos, foi criada uma nova classe `Page` no Módulo de Blocos (seção 2.3.3). As classes `Stream` e suas derivadas `PushStream` e `PullStream` foram removidas do Módulo de Metaclasses, sendo as duas últimas substituídas pelas classes `PushPage` e `PullPage`, inseridas no Módulo de Blocos. Junto com a classe `Node`, são especializações da classe `Page`, conforme mostrado pela Figura 18.

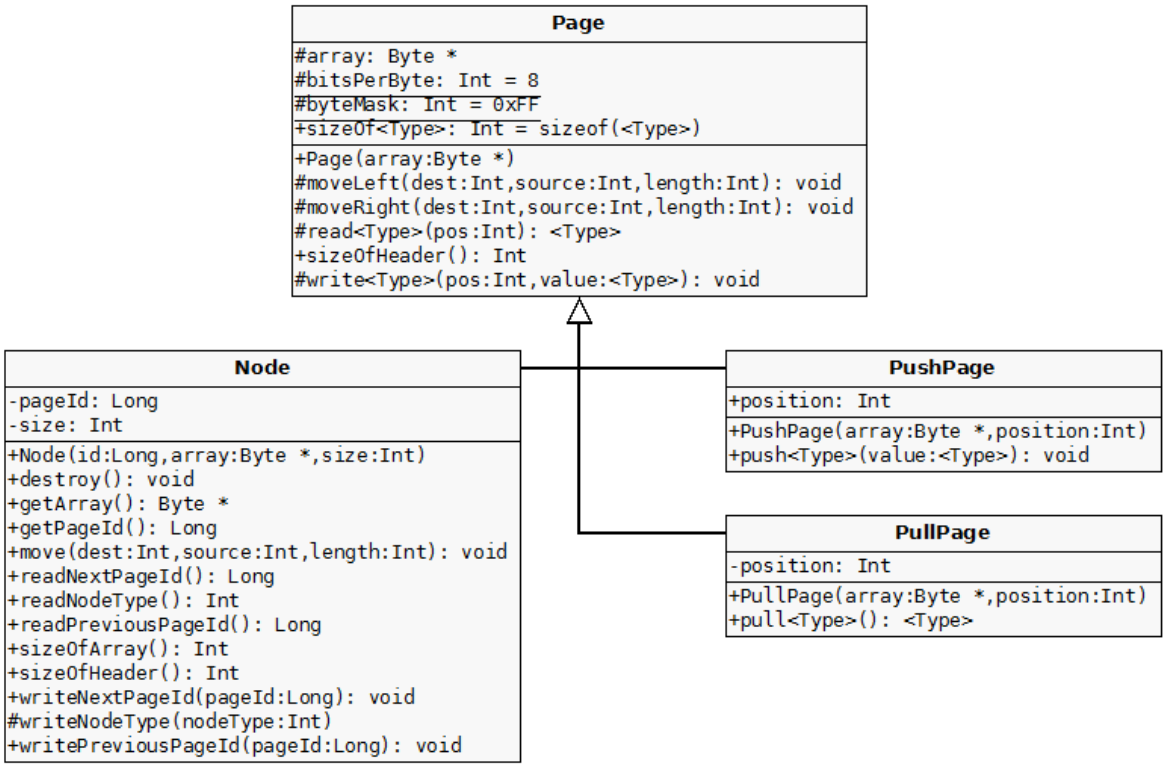


Figura 18: Modelagem da nova classe `Page` e suas derivadas no Módulo de Blocos. A classe `Page` passa a ser a única responsável pela modificação do *array* de *bytes* que reflete o conteúdo de cada bloco do *framework*, prevenindo inconsistências no compartilhamento de informações causadas por implementações diferentes.

Nesta nova hierarquia, a classe `Page` passou a ser a única responsável pela modificação do vetor de *bytes* que reflete o conteúdo de cada página do arquivo, encapsulando os méto-

dos de leitura (`read<Type>()`) e escrita (`write<Type>()`) para cada tipo primitivo. Suas classes derivadas continuam com as mesmas responsabilidades, utilizando os métodos de modificação da classe mãe.

B. Padronização da Ordenação de *Bytes* nas Operações de Serialização e Desserialização

Para garantir a compatibilidade, o modo como os dados são serializados/desserializados deve ser padronizado, permitindo que a implementação adapte a leitura e escrita de dados à arquitetura utilizada. Como o padrão da JVM é armazenar as informações que possuam mais de um *byte* em ordenação *Big-Endian* (LINDHOLM et al., 2015), optou-se por esta ordenação como padrão das operações de serialização e desserialização do *framework*.

C. Ordenação de *Bytes* em Diferentes Arquiteturas

Uma vez que o padrão de ordenação dos dados compartilhados é definido, as plataformas que utilizam esses dados devem adaptar suas operações de leitura e escrita para a ordenação de *bytes* de sua arquitetura (Seção 2.4.2). No Object-Injection, a definição da ordenação afeta diretamente a classe `Page`, responsável pelas operações de serialização e desserialização dos dados.

Para que a classe `Page` saiba qual a ordenação de *bytes* utilizada pela arquitetura, foram inseridas no arquivo `config.h` duas macros: `ENDIANESS_BIG` e `ENDIANESS_LITTLE`. Um erro de compilação é gerado se nenhuma configuração é feita ou se ambas as macros são definidas, conforme mostrado pelo Programa 9.

Programa 9: Configuração da ordenação de bytes no arquivo `config.h`.

```
/* --------- E n d i a n e s s   C o n f i g u r a t i o n --------- */
#define ENDIANESS_BIG           /*! @brief System is Big-Endian */
#define ENDIANESS_LITTLE        /*! @brief System is Little-Endian */

#if ((defined(ENDIANESS_LITTLE) && defined(ENDIANESS_BIG)) ||
  (!defined(ENDIANESS_LITTLE) && !defined(ENDIANESS_BIG)))
#error "ObI_003: Inconsistent Endianess."
#endif
```

Na classe `Page`, a recuperação de tipos inteiros com tamanho de mais de um *byte* são feitos por meio de operações de deslocamento e somas, conforme mostrado pelo Programa 10, que recupera uma variável do tipo `Short`. Esta técnica de serialização e desserialização produz os mesmos resultados independente do sistema de ordenação de *bytes* da arquitetura. Os dados dos tipo `Float` e `Double`, entretanto, não podem ser manipulados

da mesma forma, ou seja, a ordenação de *bytes* da arquitetura influencia o resultado final de uma operação de serialização ou desserialização. Assim, para esses tipos de dados, as macros de ordenação são utilizadas para determinar de que forma os dados são lidos ou escritos no array de *bytes*, conforme ilustrado pelo método `readFloat()` mostrado pelo Programa 11.

Programa 10: Leitura de dado `Short` independente da ordenação de *bytes*.

```
/*! \brief Read Short data from position \b pos in \b array. */
Short readShort(Int pos) {
    Short value = 0;
    value <<= bitsPerByte;
    value += array[pos] & byteMask;
    value <<= bitsPerByte;
    value += array[pos + 1] & byteMask;
    return value;
}
```

Programa 11: Leitura de dado `Float` com a configuração de ordenação de *bytes*.

```
/*! \brief Read Float data from position \b pos in \b array. */
Float readFloat(Int pos) {
    Float value;
    Byte * ptr = (Byte *) & value;
#if defined(ENDIANESS_LITTLE)
    ptr[0] = (Byte) array[pos + Page::sizeOfFloat - 1];
    ptr[1] = (Byte) array[pos + Page::sizeOfFloat - 2];
    ptr[2] = (Byte) array[pos + Page::sizeOfFloat - 3];
    ptr[3] = (Byte) array[pos + Page::sizeOfFloat - 4];
#elif defined(ENDIANESS_BIG)
    ptr[0] = (Byte) array[pos];
    ptr[1] = (Byte) array[pos + 1];
    ptr[2] = (Byte) array[pos + 2];
    ptr[3] = (Byte) array[pos + 3];
#endif
    return value;
}
```

As modificações na modelagem foram codificadas em ambas as implementações (C++ e Java), e o teste de persistência e recuperação de 100 mil entidades da classe `Student` (Figura 16) repetido. Nesta tentativa, a compatibilidade entre as implementações foi satisfeita, com todos as entidades e chaves (indexadas pelas estruturas `BTreeEntity` e `BTree`, respectivamente) persistidas pela implementação C++ encontradas pela implementação Java e vice-versa.

3.4 Análise das Operações de Escrita em Disco

Conforme exposto na Seção 2.1.2, quando cartões de memória do tipo *flash* são utilizados, é altamente desejável que o número de escritas e atualizações na memória física seja mínimo, de forma a aprimorar a performance do sistema e a vida útil do dispositivo. Em ambientes *desktop*, a diminuição do número de operações de disco promove a economia de recursos computacionais e consequente diminuição do consumo de energia.

O *framework* Object-Injection otimiza a quantidade de operações de leitura de dados do disco por meio de um *cache* de memória implementado na classe `Session` através de uma estrutura `Map<PageID,Node>`. Antes de carregar uma nova página do arquivo na memória, é verificado se esta página já está disponível no cache; caso negativo, a página é lida do disco e inserida na estrutura, descartando futuras leituras da mesma página na sessão.

Com o objetivo de avaliar quantas operações de escrita em disco são realizadas para cada página do arquivo, foram criadas duas classes temporárias no Módulo de Dispositivos (Seção 2.3.4): `NumWrites`, que armazena o número da página e quantas vezes ela foi escrita, e `WriteCounter`, que possui uma lista do tipo `NumWrites` e passou a ser um atributo da classe `Workspace`. A relação entre as classes é mostrada pela Figura 19. Durante a execução do programa, é criada uma lista contendo o `PageId` de todas as páginas atualizadas em disco com seus respectivos contadores. A lista pode ser recuperada através do método `printWrites()` da classe `WriteCounter`.

Além do número de operações de escrita em disco, foram feitas avaliações de tempo de execução com arquivos de tamanho de página diferentes (1kB, 2kB e 4kB), para as operações de persistência e recuperação, através do comando `time` do sistema operacional Ubuntu. Em ambas, entidades e chaves da classe `Student` (Figura 16) geradas a partir de um arquivo texto construído aleatoriamente foram persistidas/recuperadas em quantidades de ordem de grandeza diferentes, sendo realizadas trinta (30) replicações de cada caso de teste para descartar influências externas nos resultados. O *script* apresentado no Programa A.1 (Apêndice A.1) foi utilizado para executar as repetições.

Os testes foram executados em ambiente virtual executando o sistema operacional Ubuntu 12.04 e utilizando 4GB de memória RAM e dois núcleos da máquina hospedeira, de processador Intel® Core™ i7-3537U de 2.00GHz e 8,00GB de RAM executanto o sistema operacional Windows 8 Single Language.

Os resultados do número de operações de escrita em disco apresentam os blocos Ca-

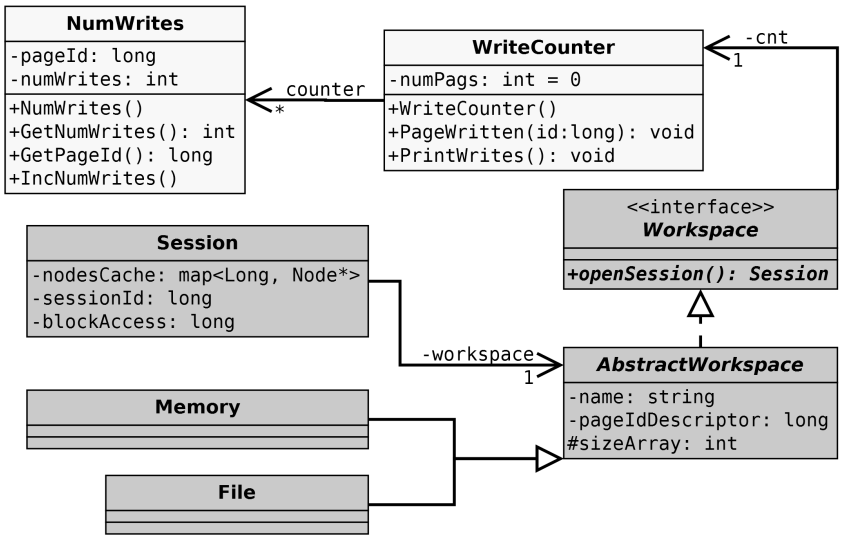


Figura 19: Modelagem das classes `WriteCounter` e `NumWrites` e sua relação com o Módulo de Dispositivos.

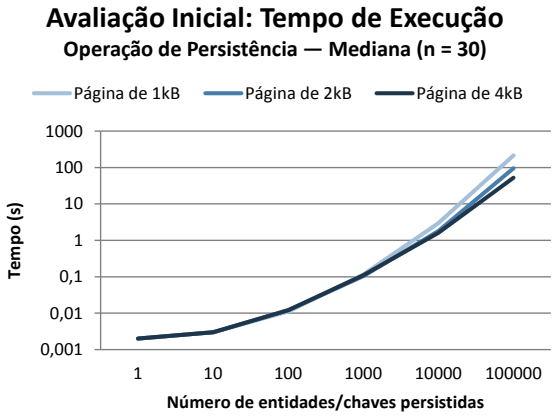
beçalho e Descritor (Seção 2.3.5) separadamente, conforme abaixo:

- **Bloco Cabeçalho** — número de vezes que o Bloco Cabeçalho foi escrito em disco;
- **Bloco Descritor** — número de vezes que o Bloco Descritor foi escrito em disco;
- **Outros Blocos** — soma do número de vezes que todos os blocos que não o Cabeçalho ou Descritor foram escritos em disco;
- **Pico de Escrita** — número de vezes que um bloco que não o Cabeçalho ou Descritor com a maior quantidade de atualizações foi escrito em disco.

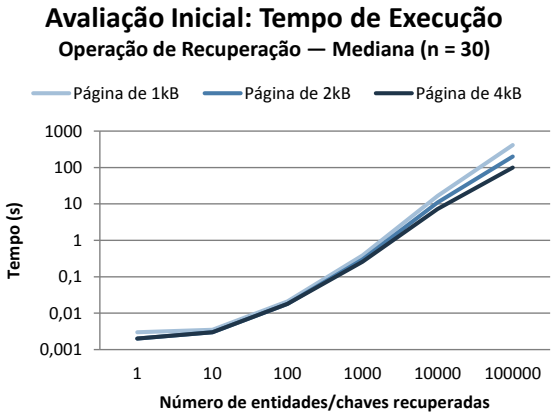
3.4.1 Avaliação Inicial

Os resultados iniciais, exibidos integralmente pelo Apêndice B.1, mostram que o tempo total de execução do *software* sofre maior influência do tamanho da página do arquivo à medida que a quantidade de entidades aumenta (Figura 20). Páginas maiores comportam mais dados, fazendo com que o número de operações em disco seja menor para ambas operações de persistência e recuperação, tornando a execução mais rápida.

Os testes apresentaram duas características em comum: (1) a proporcionalidade entre o número de operações de escrita em disco e a quantidade de entidades/chaves persistidas/recuperadas e (2) a execução de operações de escrita em disco nas operações de recuperação, as quais não modificam a estrutura de indexação. A Figura 21 exibe o número de operações de escrita em arquivo nas operações de persistência e recuperação de



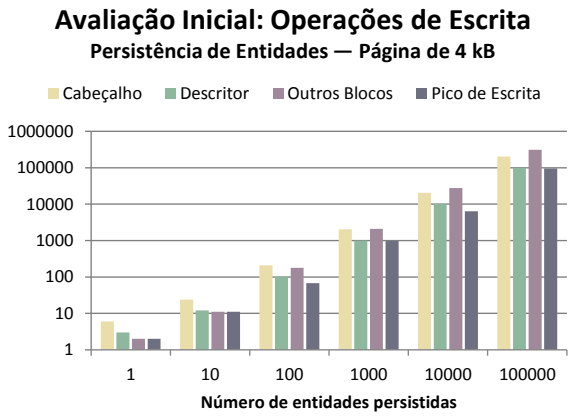
(a) Operação de Persistência



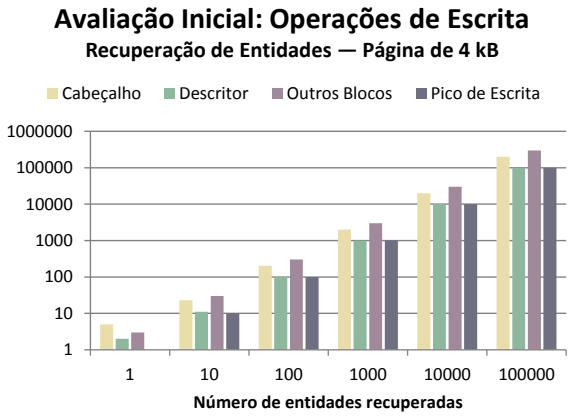
(b) Operação de Recuperação

Figura 20: Avaliação Inicial: Comparativo do tempo total de execução para diferentes tamanhos de páginas. O aumento do tamanho da página aumenta a quantidade de entidades em cada bloco, diminuindo o número de operações em disco e consequentemente o tempo de execução.

entidades em um arquivo de 4kB de página.



(a) Operação de Persistência



(b) Operação de Recuperação

Figura 21: Avaliação Inicial: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. Os blocos são atualizados em disco mesmo na execução de recuperações, quando não há modificação da estrutura de indexação.

O grande número de operações de escritas em disco na execução de recuperações ocorre porque o método `flushPage()` atualiza os blocos em disco sem verificar se ocorreram modificações em seu conteúdo. Desta forma, uma maneira de melhorar a eficiência do *software* é inserir esta verificação e, somente se atendida, atualizar o conteúdo em disco.

3.4.2 Condicionamento da Atualização em Disco

Como única responsável pela modificação do bloco que representa a página de arquivo no programa, a classe `Page` recebe a responsabilidade de controlar um *flag* que indique se aquele bloco foi modificado ou não. Assim, sua estrutura, antes vazia, passa a incluir um metadado lógico para essa indicação, denominado `Modified`, conforme mostrado pela Figura 22. O indicador, de tamanho de um *byte*, é setado todas as vezes em que a classe `Page` realiza uma operação de escrita no vetor de *bytes*.

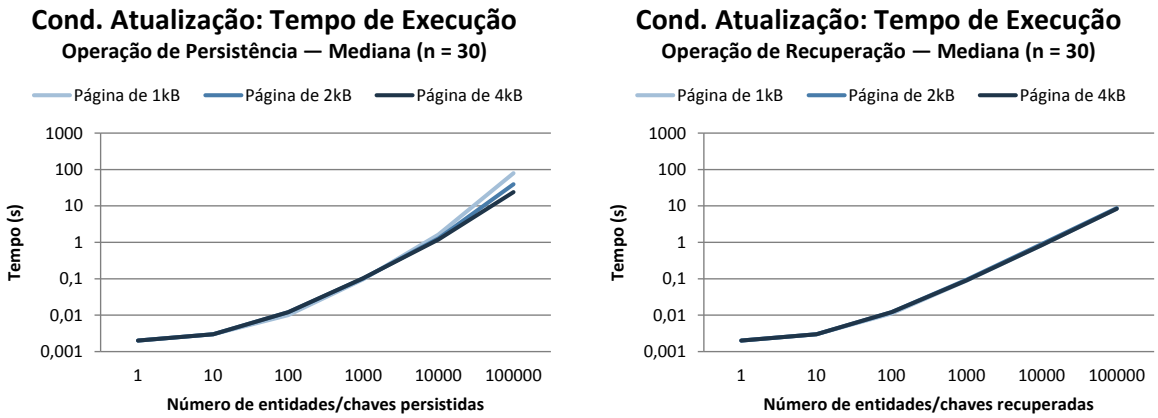
Figura 22: Metadado `Modified` adicionado à classe `Page`. O dado lógico é setado quando o bloco é modificado e verificado pelo método `flushPage()` na finalização de uma sessão.

Para condicionar a atualização dos dados do bloco em disco, foram criados dois novos métodos na classe `AbstractWorkspace` (Módulo de Dispositivos — Seção 2.3.4). O método `writePage()` realiza a operação de escrita em disco, enquanto o método `discardPage()` descarta o bloco carregado na memória. O método `flushPage()` foi modificado para testar o *flag* de modificação do bloco, invocando os métodos `writePage()` e `discardPage()` conforme necessário, conforme mostrado pelo Programa 12.

Programa 12: Modificação do método `flushPage()` na classe `AbstractWorkspace`.

```
Bool AbstractWorkspace::flushPage(Node * node){
    if(node->readModified() == true){
        node->resetModified();
        return this->writePage(node);
    }
    else{
        return this->discardPage(node);
    }
}//flushPage
```

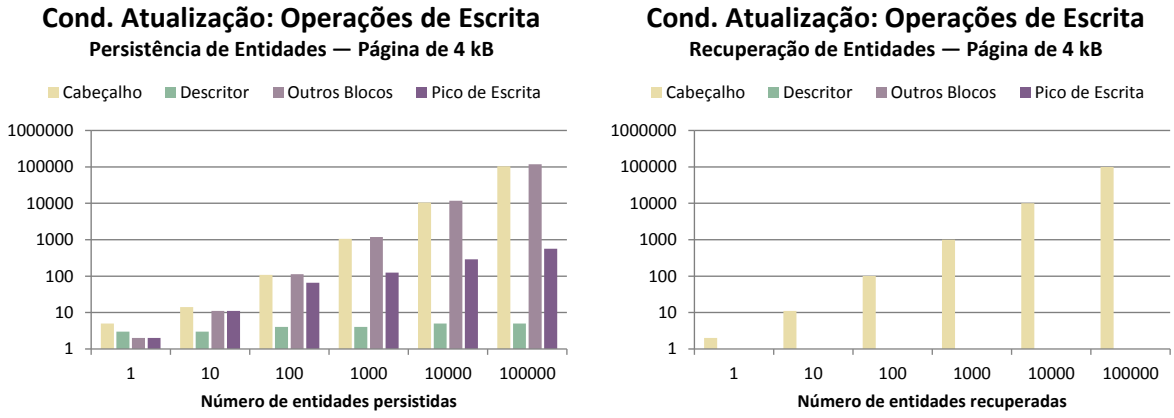
Os resultados obtidos em uma segunda execução do teste após as modificações descritas acima são exibidos integralmente pelo Apêndice B.2. É possível observar, conforme mostrado pela Figura 23, uma diminuição significativa dos tempos de execução nas operações de pesistência e recuperação em relação à implementação anterior (Figura 20).



(a) Operação de Persistência (b) Operação de Recuperação

Figura 23: Condicionamento da Atualização em Disco: Comparativo do tempo total de execução para diferentes tamanhos de páginas. Após a modificação, há uma queda significativa no tempo de execução das operações de persistência e recuperação.

A queda do número de operações de escrita em disco explicam o ganho de tempo. A Figura 24 exibe o número de operações de escrita em disco para as operações de persistência e recuperação de entidades em um arquivo de 4kB de página.



(a) Operação de Persistência (b) Operação de Recuperação

Figura 24: Condicionamento da Atualização em Disco: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. As operações de escrita em disco caem significativamente nas operações de persistência. Para as operações de recuperação, somente o Bloco Cabeçalho é atualizado devido ao registro *Last Session ID*.

Na etapa de persistência (Figura 24(a)), a atualização dos blocos em disco continua

proporcional à quantidade de entidades persistidas, porém em quantidade menor do que a encontrada na avaliação inicial (Figura 21(a)). De fato, a redução das operações de escrita na categoria *Outros Nós* chegou a 60 por cento nos casos com maior número de entidades persistidas. Tal redução é facilmente visualizada pela categoria *Pico de Escritas*, muito menor do que o obtido anteriormente, e explicada pelo fato de que, nesta nova implementação, apenas o bloco folha é atualizado quando a persistência é bem sucedida sem nenhuma operação de *split* — ao contrário do que ocorria anteriormente, onde todos os blocos carregados eram atualizados. O mesmo raciocínio se aplica às atualizações do bloco Descritor, que são feitas somente quando a altura da árvore muda.

Na etapa de recuperação (Figura 24(b)), tanto os blocos das estruturas de indexação quanto o bloco Descritor não são atualizados em disco, uma vez que não são modificados. Apenas o bloco Cabeçalho é atualizado em disco (ainda assim com redução de aproximadamente 50 por cento nos casos de maior número de objetos) devido à atualização do registro *Last Session ID* (Figura 12), incrementado toda vez que o `worspace` é acessado.

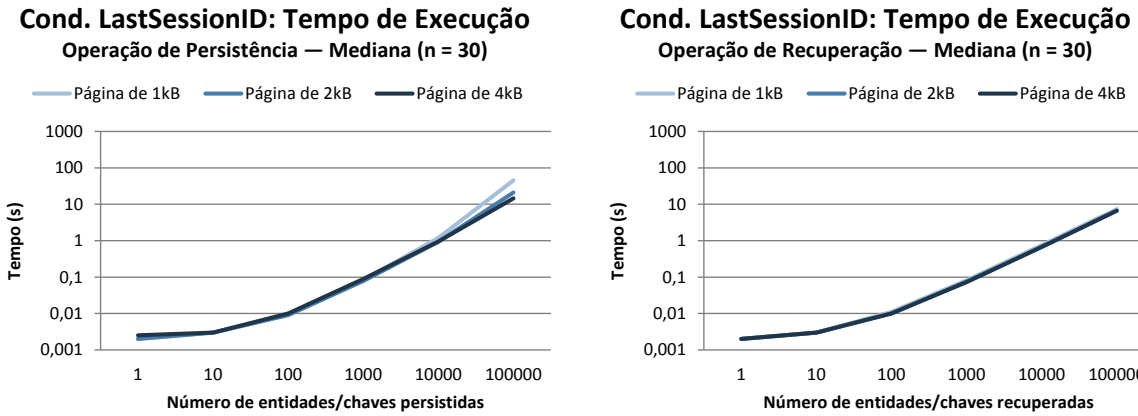
3.4.3 Condicionamento da Atualização do Registro Last Session ID

O registro *Last Session ID* é um inteiro de 64 *bits* que será utilizado para o isolamento e controle de concorrência no *framework* Object-Injection. Sua atualização no Bloco Cabeçalho pode não ser considerada relevante em uma aplicação que não necessite dessa funcionalidade. Assim, o construtor da classe `Session` no Módulo de Dispositivos (Seção 2.3.4) foi modificado conforme mostrado pelo Programa 13 para fornecer ao usuário a escolha de realizar ou não o armazenamento deste valor. O condicionamento da atualização do registro *Last Session ID* é feito através da diretiva `LOG_SESSION_ID` criada no arquivo `config.h`; o registro só será modificado se a diretiva for definida e possuir valor maior do que zero.

Programa 13: Modificação do construtor de `Session` para armazenamento opicional do campo `LastSessionID`.

```
Session::Session(AbstractWorkspace *workspace) {
    blockAccess = 0;
    this->workspace = workspace;
#if((defined LOG_SESSION_ID) && (LOG_SESSION_ID) > 0)
    sessionId = workspace->incrementSessionId();
#else
    sessionId = 0;
#endif
}
```

Os testes de verificação do número de operações de escrita em disco foram repetidos sem a definição da macro `LOG_SESSION_ID` para verificar o impacto desta opção na execução do programa. Os resultados são exibidos integralmente pelo Apêndice B.3.



(a) Operação de Persistência

(b) Operação de Recuperação

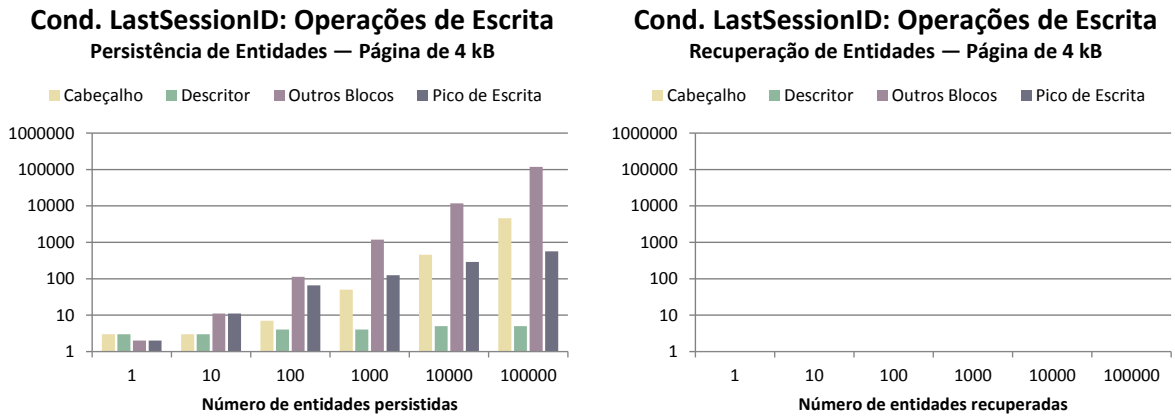
Figura 25: Condicionamento da Atualização do Registro *Last Session ID*: Comparativo do tempo de execução para diferentes tamanhos de páginas. É registrada uma pequena melhora no tempo de execução dada a diminuição do número de atualizações do Bloco Cabeçalho em disco.

Foi constatada uma pequena diminuição no tempo de execução (exibidos pela Figura 25 para uma página de tamanho de 4kB) para ambas as operações de persistência (Figura 25(a)) e recuperação (Figura 25(b)) devido à diminuição do número de escritas em disco do bloco Cabeçalho, conforme mostrado pela Figura 26, para a página de 4 kB.

Sem a atualização do registro *Last Session ID*, o bloco Cabeçalho é atualizado na criação de estruturas de indexação e quando uma nova página é inserida (registro *Last Page ID* — Seção 2.3.5). Não há diferença no número de operações de escrita em disco dos outros tipos de blocos na operação de persistência (Figura 26(a)); na operação de recuperação (Figura 26(b)), nenhum bloco foi modificado e por isso nenhuma operação de escrita em disco foi realizada.

3.4.4 Comparativo Entre as Implementações

Para melhor visualização entre os resultados obtidos nas diferentes implementações, são apresentados gráficos que comparam os resultados de tempo de execução e número de operações de escrita em disco obtidos nas três avaliações realizadas, rotuladas como *Original* (avaliação inicial), *Cond. Atualização* (condicionamento da atualização em disco)



(a) Operação de Persistência

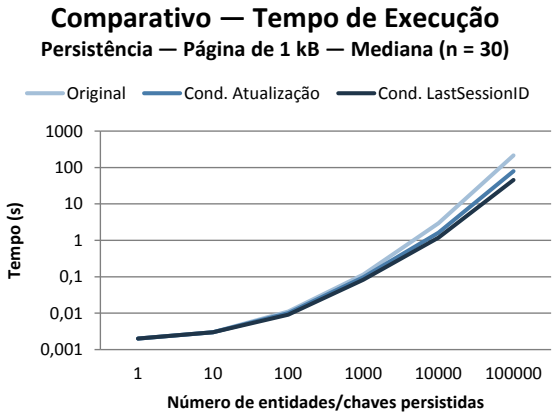
(b) Operação de Recuperação

Figura 26: Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco de diferentes blocos em arquivo de página de 4kB. O número de atualizações do Bloco Cabeçalho é reduzido consideravelmente na operação de persistência, enquanto que na operação de recuperação nenhum bloco é atualizado.

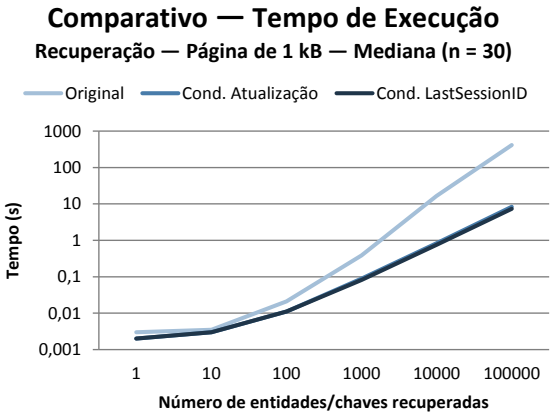
e *Cond. LastSessionID* (condicionamento da atualização do registro `LastSessionID`). A Figura 27 mostra os resulados de tempo de execução obtidos nas operações de persistência e recuperação de entidades e chaves para todos os tamanhos de página testados.

Para ambas as operações de persistência (Figuras 27(a), 27(c) e 27(e) e Tabela 3) e recuperação (Figuras 27(b), 27(d) e 27(f) e Tabela 4), é possível observar que o impacto no tempo de execução é maior à medida que o número de entidades/chaves persistidas aumenta, mas diminui com o aumento do tamanho da página. Tais resultados são explicados pela altura da árvore resultante da operação de persistência. Para um mesmo tamanho de página, a árvore resultante tem altura maior quando se persiste um maior número de entidades/chaves. Por outro lado, o aumento do tamanho da página permite a persistência de uma maior quantidades de objetos em cada bloco, tornando a árvore mais densa e consequentemente mais baixa. A altura da árvore é diretamente proporcional à quantidade de operações de leitura/escrita com o disco, impactando de forma também diretamente proporcional a quantidade de operações otimizadas. No caso de teste de 100 mil entidades/chaves persistidas/recuperadas, a redução no tempo de execução chegou a 70% na operação de persistência e ultrapassou 90% na operação de recuperação.

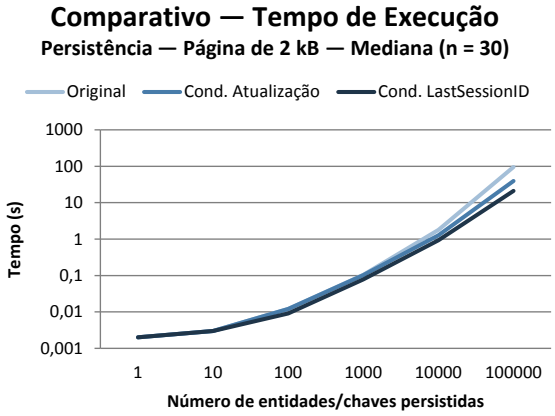
As Figuras 28 e 29 comparam, respectivamente, os resultados do número de operações de escrita em disco para as operações de persistência e recuperação de entidades em um arquivo de 4kB de página. As comparações para todos os tamanhos de página, para operações de persistência e recuperação de entidades e chaves apresentaram comportamento



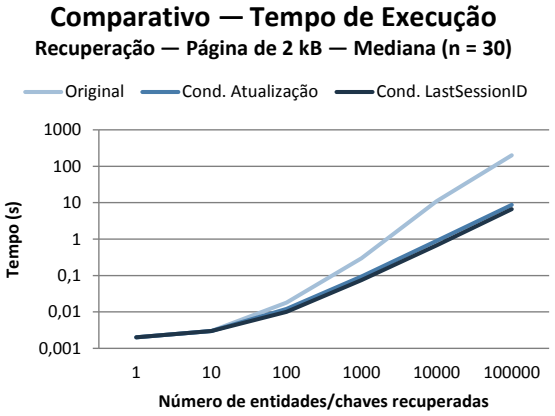
(a) Operação de Persistência — Página de 1kB



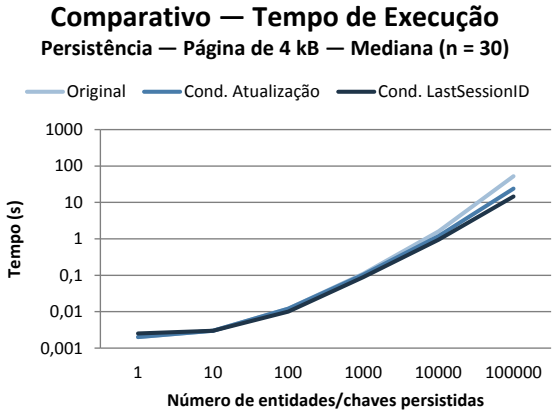
(b) Operação de Recuperação — Página de 1kB



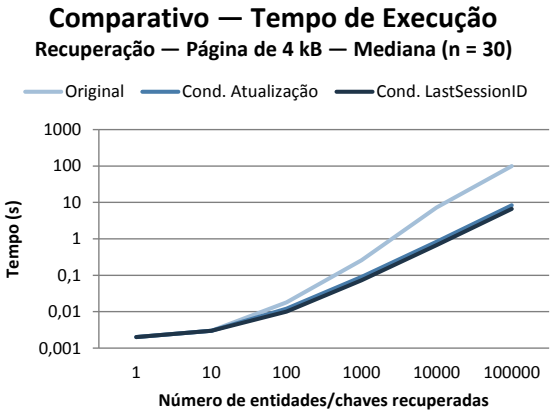
(c) Operação de Persistência — Página de 2kB



(d) Operação de Recuperação — Página de 2kB



(e) Operação de Persistência — Página de 4kB



(f) Operação de Recuperação — Página de 4kB

Figura 27: Comparativo Entre Implementações: Tempo total de execução para operações de persistência e recuperação em arquivo de 4kB de página. O condicionamento das operações de escrita, rotulado como *Cond. Atualização*, apresenta maior impacto no tempo de execução para ambos os casos.

Redução no Tempo de Execução — Operação de Persistência

Página	Nº de objetos	Tempo de Execução (s) — Mediana (n = 30)			
		Original	*Cond. Atualização*	*Cond. LastSessionID*	Redução Total* (%)
1kB	1	0,002	0,002	0,002	0,00
	10	0,003	0,003	0,003	0,00
	100	0,011	0,01	0,009	18,18
	1000	0,115	0,096	0,083	27,83
	10000	2,973	1,611	1,1795	60,33
	100000	215,186	79,8155	45,252	78,97
2kB	1	0,002	0,002	0,002	0,00
	10	0,003	0,003	0,003	0,00
	100	0,012	0,012	0,009	25,00
	1000	0,103	0,1015	0,078	24,27
	10000	1,792	1,275	0,9255	48,35
	100000	95,879	39,7545	21,2205	77,87
4kB	1	0,002	0,002	0,002	0,00
	10	0,003	0,003	0,003	0,00
	100	0,012	0,012	0,01	16,67
	1000	0,11	0,104	0,088	20,00
	10000	1,6035	1,186	0,947	40,94
	100000	52,3815	24,038	14,57	72,18

* A redução to tempo de execução é calculada entre a Avaliação Inicial (*Original*) e o Condicionamento da Atualização do Registro LastSessionID (*Cond. LastSessionID*).

Tabela 3: Redução no tempo de execução na operação de persistência devido à diminuição da quantidade de operações de escrita em disco.

semelhante.

Na operação de persistência (Figura 28), é possível observar maior impacto no número de operações de escrita em disco em todas as categorias — *Bloco Cabeçalho* (Figura 28(a)), *Bloco Descritor* (Figura 28(b)), *Outros Blocos* (Figura 28(c)) e *Pico de Escrita* (Figura 28(d)) — no condicionamento da atualização dos blocos em disco (*Cond. Atualização*). No caso do Bloco Cabeçalho, observou-se redução de aproximadamente 50 por cento no número de atualizações em disco, para ambos arquivos de persistência de entidades e chaves, independentemente do tamanho da página, a partir de 10 entidades/chaves persistidas. Para o Bloco Descritor, a queda é ainda maior, ultrapassando 96 por cento a partir de 10 entidades/chaves, para todos os tamanhos de página testados. Embora a redução do número total na categoria *Outros Blocos* seja diretamente proporcional à quantidade de entidades/chaves persistdas e dependente do tamanho da página — podendo chegar até 60 por cento — o *Pico de Escritas* apresentou redução superior a 95 por cento nos casos de maior número de entidades/chaves persistidas, para todos os tamanhos de página.

Redução no Tempo de Execução — Operação de Recuperação

Página	Nº de objetos	Tempo de Execução (s) — Mediana (n = 30)			
		Original	*Cond. Atualização*	*Cond. LastSessionID*	Redução Total* (%)
1kB	1	0,003	0,002	0,002	33,33
	10	0,0035	0,003	0,003	14,29
	100	0,021	0,011	0,011	47,62
	1000	0,3835	0,0885	0,0815	78,75
	10000	16,469	0,834	0,7525	95,43
	100000	414,5155	8,3115	7,3965	98,22
2kB	1	0,002	0,002	0,002	0,00
	10	0,003	0,003	0,003	0,00
	100	0,018	0,012	0,01	44,44
	1000	0,2965	0,094	0,075	74,70
	10000	10,929	0,888	0,668	93,89
	100000	199,7115	8,7495	6,615	96,69
4kB	1	0,002	0,002	0,002	0,00
	10	0,003	0,003	0,003	0,00
	100	0,018	0,012	0,01	33,33
	1000	0,2575	0,0905	0,073	64,85
	10000	7,2045	0,834	0,6725	88,42
	100000	99,675	8,402	6,6075	91,57

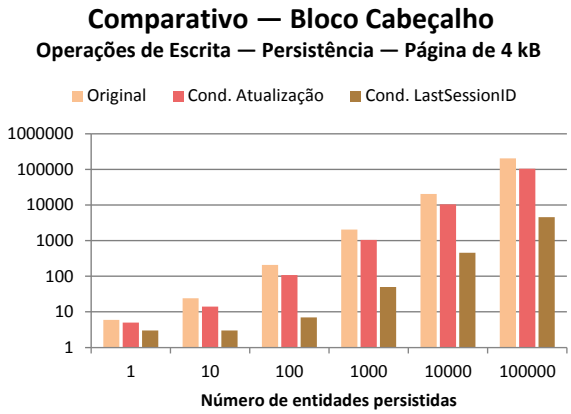
* A redução to tempo de execução é calculada entre a Avaliação Inicial (*Original*) e o Condicionamento da Atualização do Registro LastSessionID (*Cond. LastSessionID*).

Tabela 4: Redução no tempo de execução na operação de recuperação devido à diminuição da quantidade de operações de escrita em disco.

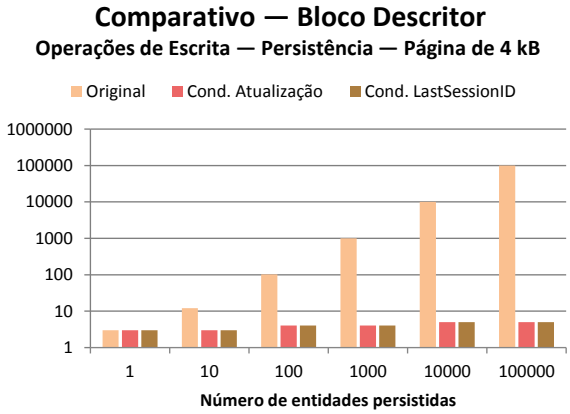
Na operação de recuperação (Figura 29), conforme refletido pelo tempo de execução, o maior impacto se dá no condicionamento da atualização do disco (*Cond. Atualização*), uma vez que só o Bloco Cabeçalho é atualizado — ainda assim com redução de 50 por cento para ambos os arquivos para os diferentes tamanhos de página. O impacto do condicionamento da atualização do registro `LastSessionID` se dá apenas nesse bloco, cujas atualizações vão a zero.

3.5 Integração do Framework Object-Injecton ao Sistema FreeRTOS™

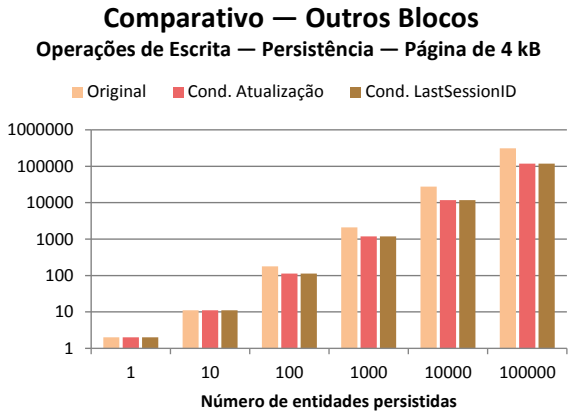
Para integrar o código do *framework* Object-Injection com o sistema FreeRTOS™, foi necessário realizar (1) a compilação conjunta de código nas linguagens C e C++, (2) a adaptação da utilização dinâmica de memória e (3) a adaptação do sistema de arquivos. Com estas ações realizadas, o *framework* Object-Injection pode ser executado por qualquer dispositivo embarcado que consiga executar o sistema FreeRTOS™ e atenda os requisitos



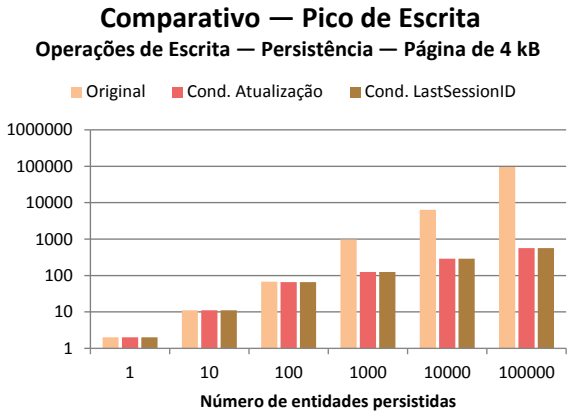
(a) Operação de Persistência — Bloco Cabeçalho



(b) Operação de Persistência — Bloco Descritor



(c) Operação de Persistência — Outros Blocos



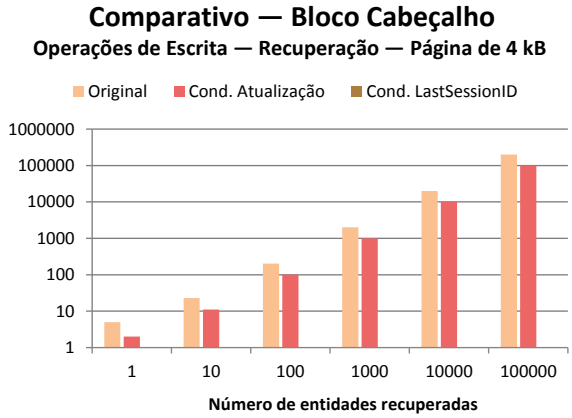
(d) Operação de Persistência — Pico de Escrita

Figura 28: Comparativo Entre Implementações: Número de operações de escrita em disco na persistência de entidades em arquivo de 4kB de página.

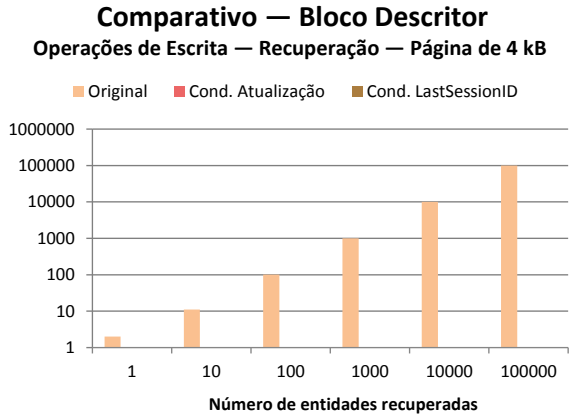
de *hardware*.

Como nesta etapa da adaptação ainda não se tem uma estimativa dos requisitos mínimos de memória do *software*, a utilização de uma plataforma embarcada pode resultar em erros causados pela insuficiência de recursos e não pela implementação em si. Para a implementação, optou-se pela utilização da versão 8.2.1 do *software* FreeRTOS™adaptado para o sistema operacional Windows (FreeRTOS™ Windows Port, 2015) utilizando o IDE Microsoft Visual Studio 2012, em conjunto com as extensões do sistema de arquivos *FAT SL* (Super Lean FAT File System, 2015) e a interface de linha de comando construída sobre um *socket* UDP (FreeRTOS™+CLI, 2015). Neste ambiente, cada tarefa é inicializada como uma *tread* do Windows; o escalonador do FreeRTOS™ apenas suspende e reinicia as *threads* — a troca de contexto é responsabilidade do sistema operacional.

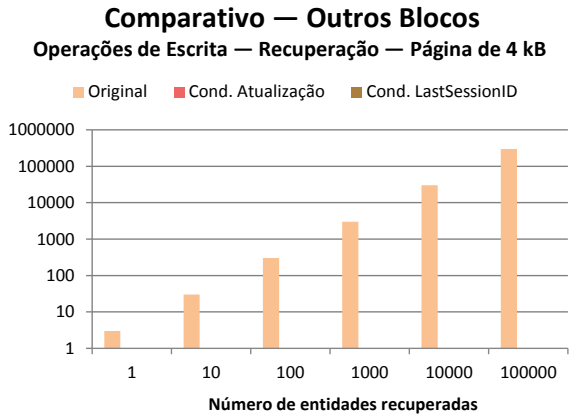
Embora este ambiente permita o desenvolvimento sem erros de memória, a aplicação



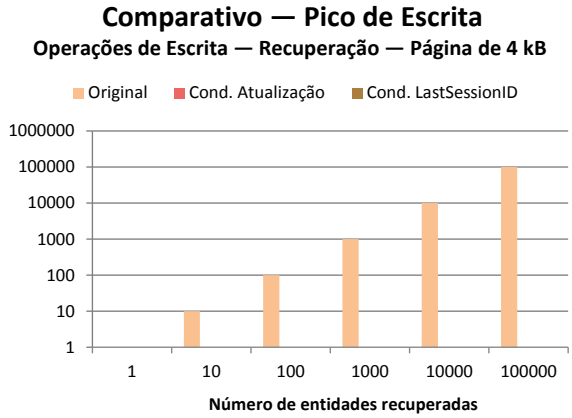
(a) Operação de Recuperação — Bloco Cabeçalho



(b) Operação de Recuperação — Bloco Descritor



(c) Operação de Recuperação — Outros Blocos



(d) Operação de Recuperação — Pico de Escrita

Figura 29: Frequência de escrita em disco na operação de recuperação de entidades em um arquivo de página de 4kB.

desenvolvida fica limitada em alguns pontos: não é possível fazer medições de tempo confiáveis, uma vez que não é garantido o comportamento de tempo real; não é possível verificar o uso da pilha de *stack* — por executar as tarefas por meio de *threads*, é utilizada a pliha do sistema operacional — e não é possível utilizar a funcionalidade de supervisão de estouro da pilha (*stack overflow*) do sistema FreeRTOS™.

3.5.1 Compilação Conjunta de Códigos nas Linguagens C e C++

Embora o IDE Microsoft Visual Studio implemente as funcionalidades do *standard* C++11 requeridas pelo *framework* na versão 2012, a macro `__cplusplus` não reflete essa conformidade como no compilador GCC. Assim, o arquivo `config.h` foi modificado (Programa 14) de forma a garantir que a versão do IDE disponibiliza as funcionalidades da linguagem C++ requeridas pelo Object-Injection — o uso da diretiva `static_assert()` e os

testes `is_base_of()` e `is_convertible()` disponíveis na funcionalidade `type_traits()`.

Programa 14: Modificação do arquivo `config.h` para garantir a conformidade da linguagem de programação na utilização do IDE Microsoft Visual Studio.

```
#if (defined _MSC_VER)          /* User is using Visual Studio. */
#if (_MSC_VER < 1700)           /* Check compiler version. */
#error "ObI_001: C++ 11 compiler required. Please use Visual Studio 2012
  or later."
#endif
#elif ((!defined __cplusplus) || (__cplusplus < 201100)) /* GCC compiler */
#error "ObI_001: C++ 11 compiler required. Please set -std=c++11 option."
#endif
```

Após modificação do arquivo, foi criada uma tarefa em um programa principal para realizar a persistência e a recuperação de objetos, ainda utilizando as funções do sistema operacional Windows para o uso dinâmico de memória e de arquivos. A compilação e execução do código foram bem sucedidas, confirmando o suporte do compilador às necessidades do software.

3.5.2 Adaptação da Utilização Dinâmica de Memória

O sistema FreeRTOS™, conforme discutido na Seção 2.1.1.1, trata a alocação dinâmica de memória como pertencente à camada portátil. Todas as operações de alocação e dealocação de memória são realizados pelos métodos `pvPortMalloc()` e `vPortFree()`. Cada dispositivo deve fornecer uma implementação para esses métodos, que possuem a mesma assinatura dos métodos `malloc()` e `free()` da biblioteca padrão da linguagem C.

Na linguagem C++, o gerenciamento da memória dinâmica é feito por meio dos operadores `new` e `delete` (DEITEL; DEITEL, 2011). A chamada de um operador `new` aloca a quantidade necessária de memória para armazenar um objeto no *heap*, encaminha a execução para o construtor apropriado e retorna um ponteiro do objeto recém-alocado. A chamada de um operador `delete` executa o destrutor daquele objeto e posteriormente libera a memória armazenada por aquele ponteiro.

Os operadores de uso dinâmico de memória — `new`, `delete`, `new[]` e `delete[]` — podem ser *sobrecarregados*. A *sobrecarga de operadores* é uma funcionalidade da linguagem C++ que permite modificar o comportamento padrão de um operador na execução do *software*. A sobrecarga pode ocorrer tanto no contexto da classe — a modificação se aplica apenas para operações com suas instâncias — quanto no contexto global — a

modificação se aplica em operações executadas em qualquer ponto do *software* (DEITEL; DEITEL, 2011).

Para sobrecarregar um operador, deve ser escrita uma função cujo nome comece com a palavra reservada `operator` seguido do operador que se deseja sobrecarregar (DEITEL; DEITEL, 2011). Assim, para que o uso dinâmico de memória seja sempre feito por meio dos métodos `pvPortMalloc()` e `vPortFree()`, foi necessário implementar a sobrecarga de operadores no contexto global do programa métodos que invoquem o uso dinâmico de memória do sistema FreeRTOS™.

Para tanto, são inseridas no arquivo principal do projeto base (arquivo `main.cpp`) as funções de sobrecarga de todos os quatro operadores de gerenciamento de memória dinâmica, conforme exposto pelo Programa 15. A implementação no arquivo principal garante que (1) a sobrecarga ocorra em um contexto global, fazendo com que qualquer operação de uso de memória dinâmica em qualquer ponto do *software* seja feita por meio das rotinas padrão do sistema FreeRTOS™; (2) a modificação fique transparente para o usuário, o qual poderá continuar utilizando os operadores `new` e `delete` em suas classes e métodos; e (3) a sobrecarga não afete o código fonte original do *framework*.

Programa 15: Sobrecarga dos operadores de gerenciamento de memória dinâmica no arquivo principal do projeto de demonstração.

```
using namespace std;

void * operator new(std::size_t size) throw (std::bad_alloc) {
  return pvPortMalloc(size);
}

void operator delete(void * pointer) {
  vPortFree(pointer);
}

void * operator new[](std::size_t size) throw (std::bad_alloc) {
  return pvPortMalloc(size);
}

void operator delete[](void * pointer) {
  vPortFree(pointer);
}
```

A sobrecarga dos operadores foi testada por meio de *breakpoints* em ambiente de depuração. Foram verificadas a criação de variáveis e vetores de tipos primitivos tanto no método principal quanto em rotinas do *framework*, bem como a criação de entidades de diferentes classes. Em todos os casos, a sobrecarga dos operadores foi executada, e os métodos `pvPortMalloc()` e `vPortFree()` invocados.

3.5.3 Adaptação do Sistema de Arquivos

O sistema FreeRTOS™, conforme discutido na Seção 2.1.1.1, possui uma extensão compatível que implementa o sistema FAT de arquivos em dispositivos *baremetal*. Para sua utilização pelo *framework* Object-Injection, é necessário substituir as invocações dos métodos de arquivo do sistema operacional por métodos da extensão FAT SL.

A. Novo Meio de Armazenamento — A classe `FATSLFile`

Para realizar a interface entre as operações de arquivo da extensão FAT SL e o *framework* Object-Injection, foi criada no Módulo de Dispositivos (Seção 2.3.4) uma nova classe `FATSLFile`, derivada da classe `AbstractWorkspace`. Nesta classe, os métodos do meio de armazenamento são implementados utilizando a API da extensão FAT SL.

Para que a compilação desta classe — e consequente inclusão de arquivos pertencentes ao FreeRTOS™ e suas extensões — não interferisse com a compilação do código original do *framework*, foi criada uma nova macro denominada `FREERTOS_COMPATIBILITY` no arquivo `config.h` para isolar quaisquer modificações necessárias para a integração do *framework* com o sistema FreeRTOS™. Todo o código da classe `FATSLFile` é condicionado a essa macro, confome mostrado pelo Programa 16.

B. Inserção de Semáforo para Acesso com Exclusão Mútua

Para garantir o isolamento das operações dos arquivos, a extensão FAT SL possui a opção de utilizar semáforos nas operações com arquivos, garantindo que apenas uma tarefa acesse o arquivo por vez. Entretanto, isso não é suficiente para o *framework*, o qual deve garantir que a tarefa seja a única a acessar o arquivo durante toda a operação de persistência ou recuperação.

Para implementar a exclusão mútua no acesso ao arquivo de persistência, é inserido na classe `FATSLFile` um atributo do tipo `xSemaphoreHandle`, utilizado pelo FreeRTOS™ para a implementação de semáforos e inicializado nos construtores da classe `FATSLFile` a partir do método `xSemaphoreCreateMutex()`. A utilização do semáforo nativo garante o isolamento das operações independente do sistema de arquivos utilizado, já que o código do *framework* não é modificado.

Para que o semáforo opere corretamente, foram realizadas algumas modificações nas classes `AbstractWorkspace` e `Session`. As implementações têm compilação condicionada pela macro `FREERTOS_COMPATIBILITY` e são discutidas a seguir.

Programa 16: Condicionamento de compilação da classe `FATSLFile`.

```
#ifndef ORG_OBINJECT_DEVICE_FATSLFILE_H
#define ORG_OBINJECT_DEVICE_FATSLFILE_H

#include <org/obinject/all.h>
#include <org/obinject/device/all.h>
#include <org/obinject/device/AbstractWorkspace.h>

#if(FREERTOS_COMPATIBILITY > 0)

#include "fat_sl.h"

using namespace std;

namespace org {
namespace obinject {
namespace device {

class FATSLFile : public AbstractWorkspace {
private:
    F_FILE * file;
    xSemaphoreHandle itemSemaphore;

  //.... Methods go here

}; //end FATSLFile

} /* device */
} /* obinject */
} /* org */

#endif /* FREERTOS_COMPATIBILITY */
#endif /* ORG_OBINJECT_DEVICE_FATSLFILE_H */
```

Primeiramente, é necessário prover métodos para a utilização do semáforo. Para tanto, foram criados na classe `AbstractWorkspace` os métodos `reserve()` e `release()`, que devem ser sobrecarregados pelas classes derivadas utilizadas com o FreeRTOS™. Os métodos devem realizar, respectivamente, as operações de obtenção e liberação do semáforo do arquivo através dos métodos `xSemaphoreTake()` e `xSemaphoreGive()` disponibilizados pelo FreeRTOS™.

Também, é necessário invocar os métodos criados para isolar o uso do recurso durante toda a interação do *framework* com o arquivo de persistência. Assim, os métodos `reserve()` e `release()` devem ser invocados, respectivamente, no início e no término de uma sessão, o que ocorre nos métodos `openSession()` na classe `AbstractWorkspace` e `close()` na classe `Session`. As modificações, expostas nos Programas 17 e 18, garantem que apenas uma sessão tenha acesso ao arquivo de persistência de cada vez.

Programa 17: Modificação do método `openSession()` para uso do semáforo na classe `AbstractWorkspace`.

```
Session *AbstractWorkspace::openSession() {
    Session * se = new Session(this);
#if(FREERTOS_COMPATIBILITY)
  if(this->reserve() == true){
    return se;
  }
  else{
    delete se;
    return NULL;
  }
#else
  return se;
#endif
}//openSession
```

Programa 18: Modificação do método `close()` para uso do semáforo na classe `Session`.

```
void Session::close() {
    map<Long, Node*>::iterator it = nodesCache.begin();
    Node *node;
    while (it != nodesCache.end()) {
        node = it->second;
        workspace->flushPage(node);
        it++;
    }
    nodesCache.clear();
#if(FREERTOS_COMPATIBILITY)
  this->workspace->release();
#endif
}
```

C. Criação de Métodos Auxiliares

Além das implementações discutidas anteriormente, foi necessário criar métodos auxiliares para a realização de testes no sistema operacional Windows. Como o disco FAT criado pela extensão é implementado como um vetor na memória RAM (o que provoca a perda de todos os arquivos criados pela aplicação quando esta é finalizada), foram criados os métodos `xLoadFile()` e `xDownloadFile()` que realizam a transferência de arquivos, respectivamente, do disco rígido para o sistema FAT SL e vice-versa.

Após as ações descritas nesta Seção, foi realizado um pequeno teste funcional com a classe `Student` (Figura 16). Foram persistidas 100 mil entidades em arquivo criado no disco de memória *heap* do sistema FreeRTOS™, o qual foi transferido para o disco rígido do computador e verificado. Também, o mesmo caso de teste foi executado no sistema operacional Windows, tendo seu arquivo resultante carregado no disco de memória *heap*

do sistema FreeRTOS™ e posteriormente utilizado para realizar operações de recuperação das 100 mil entidades. Nenhum erro foi encontrado e todos os objetos foram encontrados corretamente, confirmando o funcionamento da integração do *framework* com o sistema FreeRTOS™.

3.6 Considerações Finais

Este Capítulo descreveu as atividades realizadas por este trabalho para realizar a adaptação do *framework* Object-Injection para o sistema FreeRTOS™. Para atingir a compatibilidade entre as implementações C++ e Java do *framework*, foi necessário padronizar os tipos de dados utilizados e o modo como as implementações executavam as operações de serialização e desserialização de dados. Foram criados tipos de dados customizados na aplicação C++ correspondentes às características dos tipos primitivos da linguagem Java utilizando variáveis de tamanho fixo das linguagens C e C++. Também, as operações de serialização e desserialização de dados foram centralizadas em uma nova classe `Page`, deixando a implementação menos propensa a erros.

Foram realizadas otimizações no *framework* relacionadas à quantidade de operações de atualização de dados em disco. A atualização dos dados de cada bloco no disco passou a ser condicionada à sua modificação e, no caso específico do Bloco Cabeçalho (Seção 2.3.5), à atualização do resgistro *Last Session ID*. As análises realizadas mostraram que a redução de operações de escrita impacta diretamente o tempo de execução do *software*, o qual sofreu redução de mais de 70% para as operações de persistência e de mais de 90% nas operações de recuperação, no caso de 100 mil entidades/chaves persistidas/recuperadas.

A adaptação final do *framework* Object-Injection para o sistema FreeRTOS™ incluiu a compilação conjunta do código C++ do *framework* e o código C do FreeRTOS™, além da utilização dos métodos de uso de memória dinâmica e de gerenciamento de arquivos do FreeRTOS™. A sobrecarga dos operadores `new`, `new[]`, `delete` e `delete[]` transferiu o gerenciamento da memória dinâmica para o sistema FreeRTOS™, enquanto que a criação de um novo meio de armazenamento através da classe `FATSLFile` permitiu a utilização dos métodos de manipulação de arquivos da extensão FATSL do sistema FreeRTOS™.

Com a adaptação concluída, foram realizados experimentos, mostrados pelo Capítulo 4, para comprovar a compatibilização das implementações e a portabilidade do *software* para outras plataformas, além do funcionamento da adaptação do *framework* para o sistema FreeRTOS™.

4 Experimentos

Este Capítulo apresenta os experimentos realizados para avaliar a implementação na linguagem C++ do *framework* Object-Injection e sua adaptação para sistemas embarcados. Foram realizados três experimentos, com objetivos distintos. O Teste de Compatibilidade (Seção 4.1) visou comprovar a compatibilidade entre as implementações C++ e Java do *framework* e a portabilidade do *software* para outra plataforma de *hardware*; o Teste Estimativo (Seção 4.2) visou avaliar o uso de memória *heap* utilizada pelo conjunto Object-Injection + FreeRTOS™ em diferentes casos de tamanho da entidade e quantidade de estruturas utilizadas; por fim, o Teste Funcional (Seção 4.3) objetivou comprovar o funcionamento do *framework* com o sistema FreeRTOS™ em um pequeno exemplo de aplicação.

4.1 Teste de Compatibilidade

O objetivo deste experimento foi comprovar a compatibilidade entre as linguagens de implementação C++ e Java do *framework* Object-Injection e verificar a portabilidade da aplicação C++ para outra plataforma de *hardware*.

A compatibilidade entre linguagens no *framework* Object-Injection é comprovada quando as operações de persistência e recuperação são realizadas de forma transparente entre as implementações de linguagens diferentes, ou seja, os dados persistidos em uma aplicação desenvolvida em uma linguagem devem ser recuperados com sucesso por uma aplicação desenvolvida em outra linguagem, validando o compartilhamento de informações. A portabilidade entre plataformas é medida a partir das adaptações necessárias para que um *software* seja transportado com sucesso de uma plataforma para outra.

A base de dados deste experimento foi extraída do banco de dados público de boletins de ocorrência (exceto assassinatos) da cidade de Chicago (Chicago Police Department — Crime Reports — 2001 to present, 2015). A base foi escolhida devido à diversidade dos seus campos,

o que permitiu a criação de uma classe com atributos de tipos variados. Devido à grande quantidade de registros disponíveis, foi utilizado um subconjunto da base, correspondente a todos os boletins de ocorrência do ano de 2014. O filtro pode ser feito no próprio *website* e exportado. Alguns dos campos foram selecionados para a construção do exemplo:

- **ID**: identificador do registro, codificado como `Long`;
- **Case Number**: identificador do caso, representado por uma `string`;
- **Block**: endereço aproximado da ocorrência, representado por uma `string`;
- **IUCR**: código do *Illinois Uniform Crime Reporting* (Chicago Police Department — Illinois Uniform Crime Reporting (IUCR) Codes, 2015) utilizado para a descrição do crime, representado por uma `string`;
- **Location Descriptor**: descreve o local do crime (como por exemplo beco, rua, apartamento, entre outros), codificado como `Short`;
- **Arrest**: atesta se ocorreu ou não uma prisão, codificado como `Bool`;
- **Community Area**: em qual área da cidade ocorreu o crime, codificado como `Byte`;
- **X Coordinate** e **Y Coordinate**: coordenadas X e Y do local do crime, codificadas como `Long`;
- **Latitude** e **Longitude**: coordenadas do local do crime, codificadas como `Double`.

Os dados acima foram isolados da base de dados utilizando o código R ilustrado pelo Programa 20 (Apêndice A.2) e utilizados na preparação de um arquivo texto com seus valores e um UUID para a realização dos testes. O UUID foi armazenado em arquivo (ao invés de gerá-lo automaticamente) para que seja utilizado no teste de recuperação, tornando possível comprovar a recuperação correta de seus atributos.

A classe `CrimeRecord` foi então implementada nas linguagens C++ e Java, juntamente com uma classe para a persistência de entidades (classe `EntityCrimeRecord`) e três classes para indexar chaves: (1) `OrderCrimeRecord`, para indexação dos registros pelo atrributo `ID` em uma Árvore B; (2) `PointCrimeRecord`, para indexação dos registros pelos atributos `X Coordinate` e `Y Coordinate` em uma Árvore M; e (3) `RectangleCrimeRecord`, para indexação dos registros pelos atributos `Latitude` e `Longitude`. A Figura 30 apresenta a classe `CrimeRecord`, suas derivadas e relações com o Módulo de Metaclasses (Seção 2.3.1). Também foram construídas aplicações (arquivos executáveis) nas linguagens C++ e Java para realizar a persistência e recuperação de todas as entidades e chaves presentes no arquivo texto.

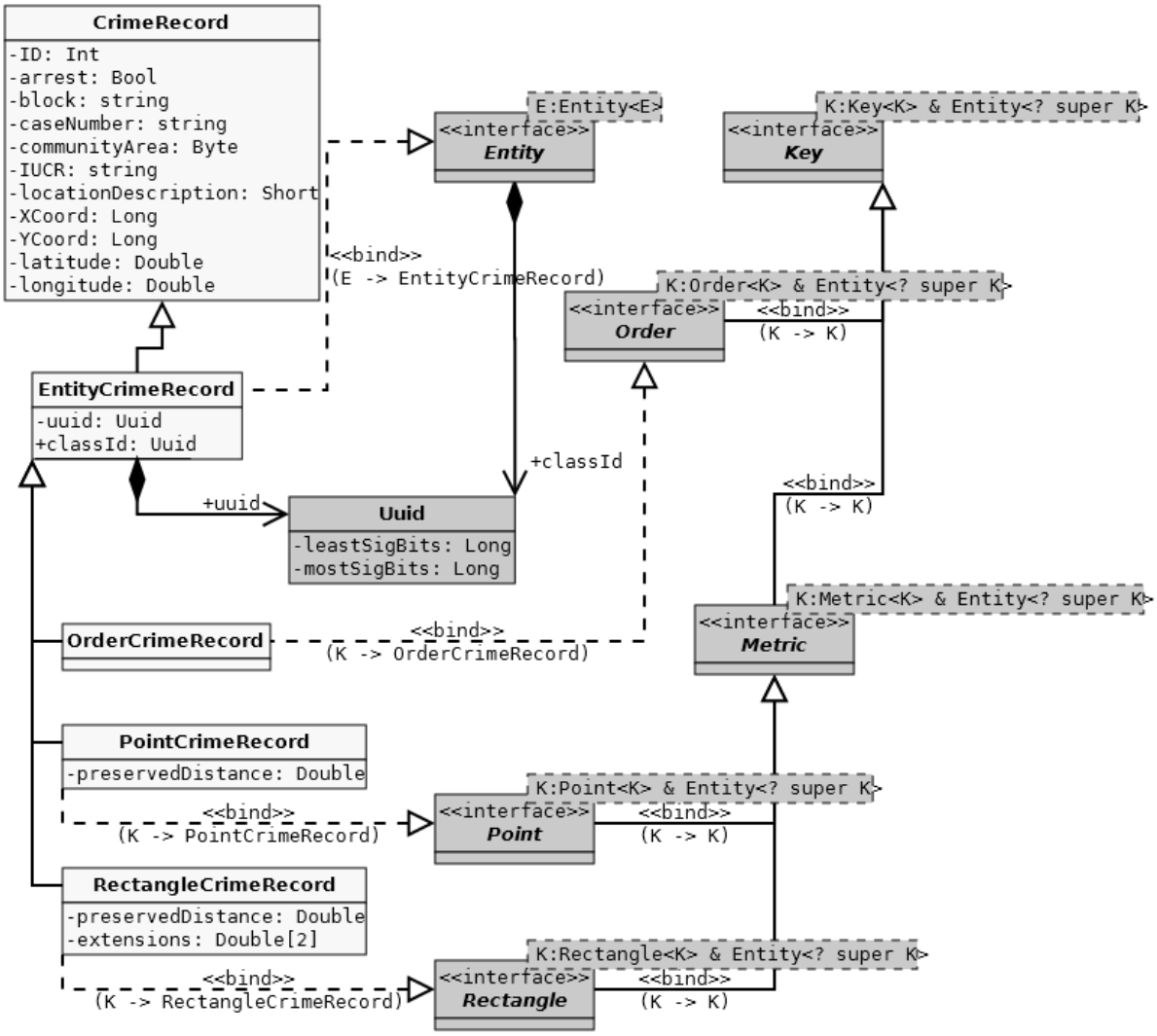


Figura 30: Classe `CrimeRecord`, suas derivadas e relações com o Módulo de Metaclasses.

O experimento foi organizado em duas etapas. A primeira etapa, concentrada na compatibilidade entre as linguagens, foi divida em dois passos, conforme mostrado pela Figura 31. No primeiro passo, foram executadas as aplicações de persistência implementadas em C++ e Java, gerando, respectivamente, os arquivos de persistência `ObI_C++` e `ObI_Java`. Para garantir a correta construção dos arquivos, também foram executadas as aplicações de recuperação na mesma linguagem utilizada para a persistência. Os arquivos gerados foram utilizados no segundo passo do teste, que consistiu na execução das aplicações de recuperação em linguagem diferente daquela utilizada na persistência.

A execução da primeira etapa foi realizada em um máquina Intel® Core™ i7-3537U de 2.00GHz e 8,00GB de RAM executanto o sistema operacional Windows 8 Single Language. Cada um dos passos foi repetido 30 vezes para validade estatística, utilizando o *script*

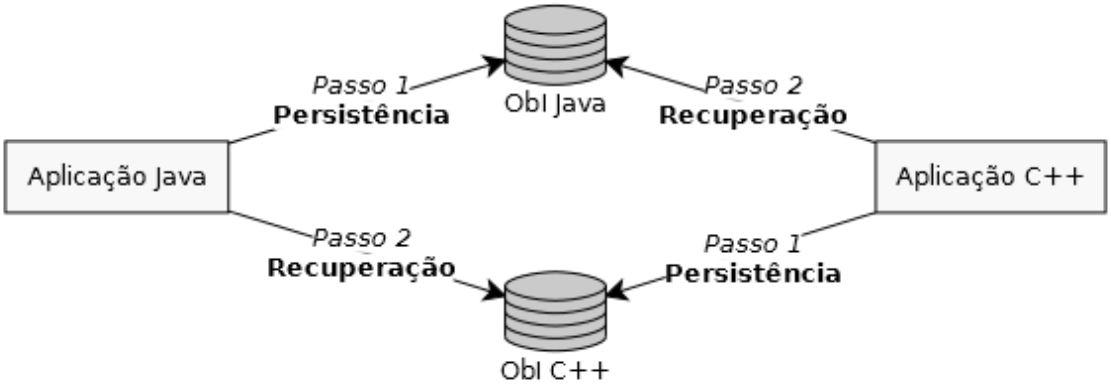


Figura 31: Teste de Compatibilidade: Primeira etapa. No primeiro passo, foi feita a persistência de objetos em uma das linguagens C++ ou Java; os arquivos de persistência gerados (`ObI_C++` e `ObI_Java`) foram utilizados no segundo passo, que realizou a recuperação dos objetos pela aplicação em uma linguagem diferente daquela utilizada na persistência.

mostrado pelo Programa 21 (Apêndice A.2).

A segunda etapa se concentrou na portabilidade da implementação C++ do *framework* e utilizou a mesma aplicação (projeto) preparada para ambiente *desktop* da etapa anterior, mas compilada para uma plataforma embarcada[1]. Esta etapa também foi dividida em dois passos, conforme mostrado pela Figura 32. No primeiro passo, foi executada a aplicação de persistência implementada em C++ na plataforma embarcada, gerando o arquivo de persistência `ObI_BBB`. No segundo passo, foi realizada a troca de arquivos entre as plataformas — a plataforma embarcada executou a aplicação de recuperação para os arquivos de persistência gerados pelas aplicações nas linguagens C++ e Java, e as aplicações de recuperação C++ e Java foram executadas para o arquivo de persistência gerado pela plataforma embarcada.

A execução da segunda etapa foi realizada na mesma máquina da primeira etapa e em uma plataforma de desenvolvimento BeagleBone Black (`http://beagleboard.org/black`), que inclui um processador AM335x 1GHz ARM® Cortex-A8 com 512MB de memória RAM e 4GB de memória *flash*, executando a distribuição do sistema operacional Ubuntu 13.10 (Saucy Salamander) já compilada para a plataforma (`http://elinux.org/BeagleBoardUbuntu`). Cada passo foi repetido 30 vezes, utilizando os *scripts* mostrados pelos Programas 22 (primeiro passo), 23 e 24 (segundo passo), incluídos no Apêndice A.2.

Para cada operação de persistência ou recuperação, foram extraídas três informações:

[1] O mesmo projeto no IDE NetBeans foi compilado utilizando um servidor remoto da plataforma embarcada.

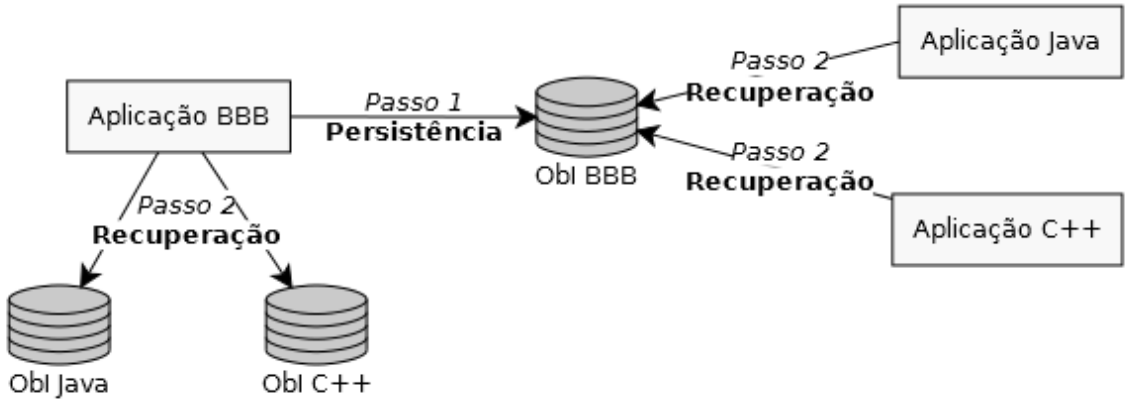


Figura 32: Teste de Compatibilidade: Segunda etapa. No primeiro passo, foi feita a persistência de objetos na plataforma embarcada *BeagleBone Black*; o arquivo de persistência gerado (`ObIBBB`) foi utilizado no segundo passo para a recuperação dos objetos pelas aplicações C++ e Java. Também no segundo passo, foi executada a aplicação de recuperação da plataforma *BeagleBone Black* nos arquivos de persistência gerados na etapa anterior (`ObI_C++` e `ObI_Java`).

(1) a quantidade média de verificações executadas[2]; (2) o número médio de acessos a blocos da estrutura[3]; e (3) o tempo médio de cada operação de persistência/recuperação. O conjunto de testes foi realizado para três tamanhos de página diferentes do arquivo de persistência (1kB, 2kB e 4kB) para analisar o comportamento do software em árvores de alturas variadas.

Os resultados do experimento não apresentaram nenhum erro na recuperação de entidades e chaves em nenhum caso, comprovando a compatibilidade entre as linguagens C++ e Java e também a alta portabilidade da implementação C++, já que não foi necessária nenhuma modificação no *software*, além da recompilação, para que o transporte de uma plataforma a outra fosse realizado com sucesso.

Em relação aos dados obtidos durante o experimento, foi observado um padrão para todos os tamanhos de página de arquivo testados. As Tabelas 5, 6 e 7 apresentam, respectivamente, os resultados obtidos para as páginas de 1kB, 2kB e 4kB, exibindo, para cada estrutura, a altura da árvore e, para as operações de persistência e recuperação, o número médio de verificações e de acessos aos blocos da estrutura, além do tempo médio de cada operação.

[2] O contador de verificações é incrementado sempre que há uma comparação entre entidades, chaves ou UUID's.

[3] O número de acessos a blocos da estrutura é diferente do número de leituras de páginas de disco obtidas no teste descrito pela Seção 3.4. Aqui, também é considerado o acesso a páginas que já estejam carregadas no *cache* da sessão (Classe `Session` no Módulo de Dispositivos — Seção 2.3.4).

Árvore B — Indexação de entidades (BTreeEntity)

Impl.	**Altura**	**Persistência**			**Recuperação**		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	4	31,443	5,828	57,846	33,876	5	47,336
Java		31,443	5,828	47,086	30,227	5	34,373
C++ BBB		31,443	5,828	522,521	33,876	5	395,817

Árvore B — Indexação de chaves (BTree)

Impl.	**Altura**	**Persistência**			**Recuperação**		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	4	32,103	5,071	55,4617	30,996	5	49,624
Java		32,103	5,071	39,3243	30,996	5	34,239
C++ BBB		32,103	5,071	512,946	30,996	5	420,250

Árvore M — Indexação de chaves (MTree)

Impl.	**Altura**	**Persistência**			**Recuperação**		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	5	53,419	6,033	89,241	191,341	54,061	587,588
Java		53,395	6,033	45,5541	100,723	22,472	135,847
C++ BBB		53,290	6,027	958,358	189,086	53,842	4512,351

Árvore R — Indexação de chaves (RTree)

Impl.	**Altura**	**Persistência**			**Recuperação**		
		Comparações	*Acessos*	*Tempo*	*Comparações*	*Acessos*	*Tempo*
C++ *desktop*	5	69,242	5,811	134,1859	59,381	6,206	83,868
Java		69,242	5,811	65,9543	58,321	6,132	47,099
C++ BBB		69,242	5,811	1798,5687	59,381	6,206	734,252

Tabela 5: Dados obtidos no Teste de Compatibilidade — Página de 1k.

Árvore B — Indexação de entidades (BTreeEntity)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	4	31,300	4,987	58,3308	34,838	5	52,341
Java		31,300	4,987	46,7158	29,669	5	38,236
C++ BBB		31,300	4,987	501,2782	34,838	5	404,199

Árvore B — Indexação de chaves (BTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	3	46,569	4,067	62,7205	28,186	4	41,067
Java		46,569	4,067	39,033	28,186	4	30,146
C++ BBB		46,569	4,067	575,875	28,186	4	355,294

Árvore M — Indexação de chaves (MTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	4	68,932	4,930	92,5371	176,770	34,258	430,631
Java		69,212	4,921	43,3576	95,854	13,649	94,642
C++ BBB		68,912	4,926	921,6541	156,101	31,157	3022,957

Árvore R — Indexação de chaves (RTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações	*Acessos*	*Tempo*	*Comparações*	*Acessos*	*Tempo*
C++ *desktop*	4	106,175	4,825	1910,0450	80,446	5,115	85,778
Java		106,175	4,825	62,4454	79,434	5,084	44,057
C++ BBB		106,175	4,825	1910,045	80,446	5,115	816,697

Tabela 6: Dados obtidos no Teste de Compatibilidade — Página de 2k.

Árvore B — Indexação de entidades (BTreeEntity)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	3	34,966	4,140	62,522	34,269	4	45,512
Java		34,966	4,140	48,925	27,445	4	36,903
C++ BBB*		34,966	4,140	545,763	34,269	4	376,938

* BeagleBone Black.

Árvore B — Indexação de chaves (BTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	3	81,268	3,941	91,107	28,910	4	41,855
Java		81,268	3,941	47,519	28,910	4	33,050
C++ BBB		81,268	3,941	840,198	28,910	4	358,099

Árvore M — Indexação de chaves (MTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações (n)	*Acessos* (n)	*Tempo* (μs)	*Comparações* (n)	*Acessos* (n)	*Tempo* (μs)
C++ *desktop*	3	89,464	4,033	102,330	205,822	25,013	397,619
Java		89,304	4,033	46,151	117,073	8,431	75,328
C++ BBB		89,370	4,034	1069,234	173,864	21,030	2621,063

Árvore R — Indexação de chaves (RTree)

Impl.	Altura	Persistência			Recuperação		
		Comparações	*Acessos*	*Tempo*	*Comparações*	*Acessos*	*Tempo*
C++ *desktop*	3	155,984	4,030	183,060	112,397	4,035	104,361
Java		155,984	4,030	68,6818	111,942	4,028	49,666
C++ BBB		155,984	4,030	2283,0177	112,397	4,035	967,499

Tabela 7: Dados obtidos no Teste de Compatibilidade — Página de 4k.

Para as operações de persistência, os resultados das diferentes implementações apresentaram os mesmos valores médios na quantidade de verificações e no número de acessos a páginas de arquivos para as estruturas de Árvore B (de indexação de entidades e chaves) e Árvore R, variando apenas em seu tempo médio de execução. Neste aspecto, a implementação na linguagem Java teve o menor tempo médio, já que não é necessário realizar operações de dealocação de memória devido ao *garbage collector* presente na JVM. Como

esperado, a implementação na BeagleBoard Black teve o maior tempo médio de operação, já que o processador utilizado é significativamente mais lento do que o do sistema *desktop*. Os resultados obtidos para a Árvore M foram diferentes para todas as implementações, devido ao fato de seus algoritmos utilizarem números randômicos na execução das operações de *split*, gerando árvores diferentes em cada execução.

Nas operações de recuperação, os valores médios na quantidade de verificações foi o mesmo para as implementações C++ em ambas as plataformas (*desktop* e embarcada) para todas as estruturas exceto a Árvore M (devido à diferença entre as árvores geradas). A implementação Java apresentou valor igual na estrutura de Árvore B que indexa chaves e menor nos demais casos, sugerindo maior eficiência do seu algoritmo. Conforme esperado, os valores médios de acesso a páginas de arquivo foram os mesmos para todas as implementações nas estruturas de Árvore B, uma vez que dependem somente de sua altura. Para a Árvore M, os resultados são diferentes para todas as implementações, dada a diversidade das árvores geradas. O tempo médio das operações apresentou a mesma característica das operações de busca, ou seja, as implementações em Java apresentaram menor tempo médio para execução de busca, e a implementação na plataforma embarcada apresentou o maior tempo médio.

4.2 Teste Estimativo

A estimativa de utilização de recursos é uma das principais questões no início do desenvolvimento de um sistema embarcado, uma vez que influencia a escolha do *hardware* necessário para a implementação, impactando os custos. Um valor superestimado causa subutilização de recursos da plataforma embarcada, o que encarece o projeto, enquanto um valor subestimado torna necessária a aquisição ou construção de uma segunda plataforma para o desenvolvimento e consequente adaptação do *software* desenvolvido, aumentando os custos e ainda atrasando o cronograma de execução.

O objetivo deste experimento foi obter uma estimativa do consumo de memória *heap* do conjunto Object-Injection + FreeRTOS™ dependendo do tamanho da entidade a ser persistida, a quantidade de estruturas utilizadas para indexação e o tamanho da página do arquivo de persistência.

A base de dados utilizada neste experimento foi o histograma de cores (*Color Histogram*) da base de dados *Corel Image Features Data Set* (Corel Image Features Data Set, 2015). A base apresenta o histograma HSV de 32 dimensões (com 8 intervalos de matiz e

4 de saturação) de mais de 60 mil imagens. Tais dados foram combinados para gerar novos histogramas menores, variando o tamanho da entidade. Os valores foram codificados como um vetor do tipo `Float`. A Tabela 8 exibe as dimensões e tamanhos de entidades utilizadas.

Nº de dimensões			Tamanho
Matiz	Saturação	Total	(*bytes*)
8	4	32	128
4	4	16	64
2	4	8	32
2	2	4	16

Tabela 8: Dimensões e tamanhos para cada entidade utilizada no Teste Estimativo. As dimensões de matiz e saturação do histograma original foram recombinadas de forma a criar entidades de tamanhos diferentes.

Foram implementadas a classe `ImageHistogram` e duas classes derivadas para indexação de chaves: `PointImageHistrogram`, que indexa os histogramas em uma Árvore M, e `RectangleImageHistogram`, que indexa os dados em uma Árvore R. A Figura 33 apresenta a classe `ImageHistogram`, suas derivadas e relações com o Módulo de Metaclasses (Seção 2.3.1).

Uma vez criadas as classes, foi construído um método para o teste no projeto de demonstração construído para o emulador do sistema operacional Windows utilizado por este trabalho (Seção 3.5). O pico de uso de memória *heap* é obtido através do método `xPortGetMinimumEverFreeHeapSize()` disponibilizado pelo sistema FreeRTOS™. A execução do método criado para o teste não foi feita através da criação de uma tarefa, mas sim por invocação direta antes do início do escalonador do sistema FreeRTOS™ para que não houvessem interrupções em sua execução.

Foram criadas aplicações (arquivos executáveis) para a realização de operações de persistência e recuperação da entidade, variando (1) sua dimensão, conforme mostrado pela Tabela 8; (2) a quantidade de itens persistidos (1k, 4k, 8k ou 16k registros); (3) o tamanho da página do arquivo de persistência (512B, 1kB, 2kB ou 4kB) e (4) a quantidade de estruturas utilizadas. Neste último item, foram feitas três variações, utilizando: (1) apenas uma Árvore B; (2) Árvores B e M; (3) Árvores B, M e R. Em todas as estruturas, são serializados todos os componentes do vetor que representa o histrograma.

A classe utilizada para a indexação em uma Árvore R (`RectangleImageHistogram`) possui o atributo `extensions`, um vetor do tipo `Double` utilizado para o cálculo do MBR

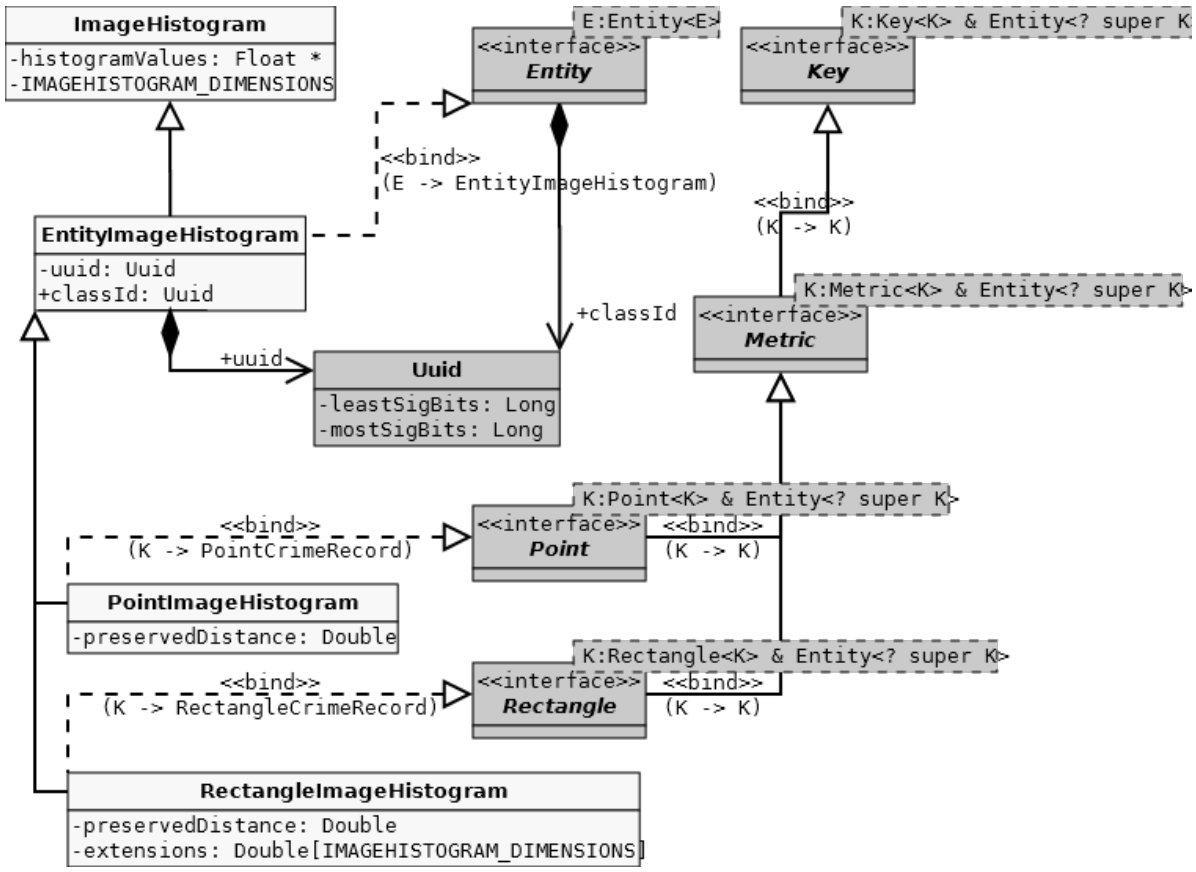
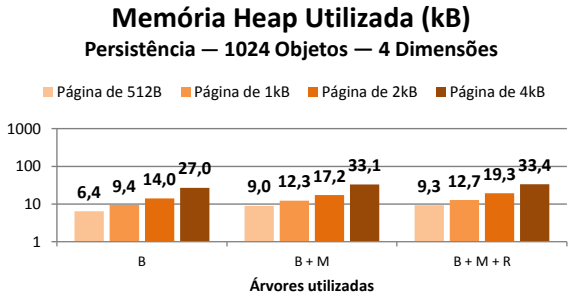


Figura 33: Classe `ImageHistogram` e derivadas e suas relações com o Módulo de Metaclasses. O histograma HSV é implementado como um vetor `Float` dinâmico de tamanho `IMAGEHISTOGRAM_DIMENSIONS`.

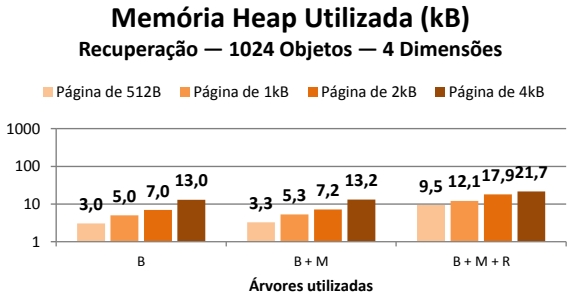
na indexação e cujo tamanho é dependente da dimensão do histograma. No caso de 32 dimensões, o tamanho do objeto indexado pela Árvore R é de 384 *bytes* (32 dimensões do tipo `Float` + 32 dimensões do vetor `Double`), o que impede a sua indexação em um arquivo de 512B de página — são necessárias pelo menos duas entidades em uma página para que a indexação ocorra. Assim, no teste de página de arquivo de 512B a combinação Árvore B + Árvore M + Árvore R não foi executada.

A execução foi realizada em uma máquina Intel® Core™ i7-3537U de 2.00GHz e 8,00GB de RAM executanto o sistema operacional Windows 8 Single Language. Cada teste foi repetido 30 vezes, utilizando o *script* mostrado pelo Programa 25 (Apêndice A.3).

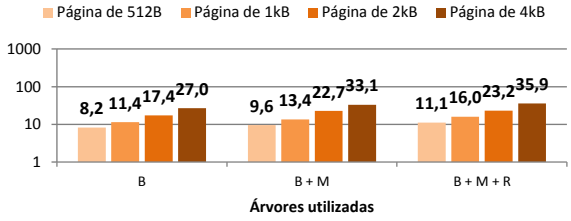
Os resultados obtidos para as entidades de 4, 8, 16 e 32 dimensões são exibidos, respectivamente, pelas Figuras 34, 35, 36 e 37. É possível observar que o aumento do tamanho da entidade (número de dimensões) e da quantidade de objetos aumenta a quantidade de memória *heap* utilizada.



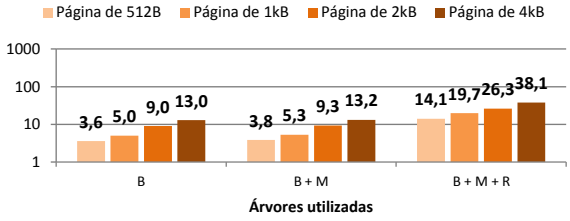
(a) Operação de Persistência — 1k entidades.



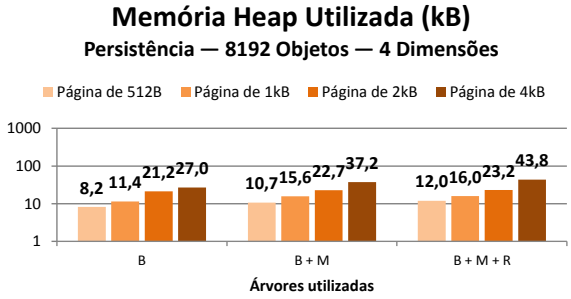
(b) Operação de Recuperação — 1k entidades.



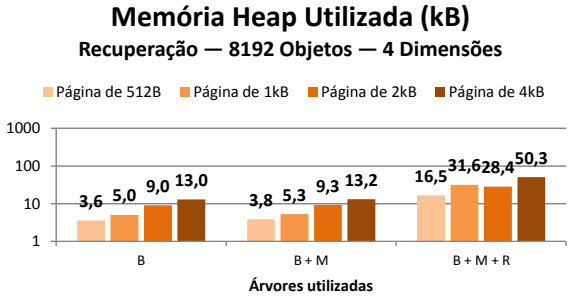
(c) Operação de Persistência — 4k entidades.



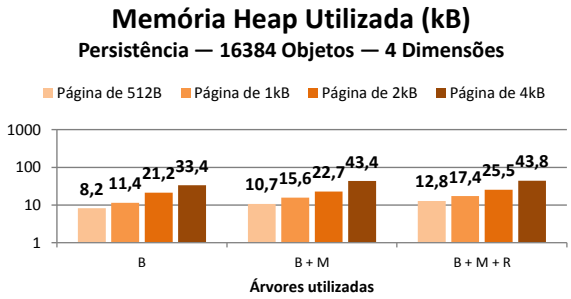
(d) Operação de Recuperação — 4k entidades.



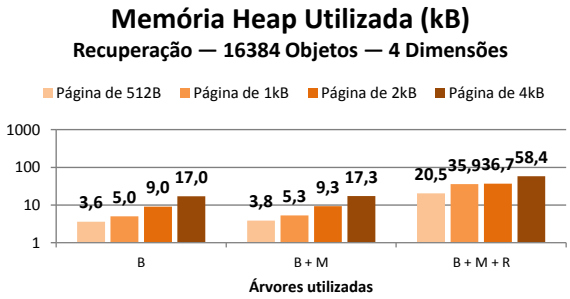
(e) Operação de Persistência — 8k entidades.



(f) Operação de Recuperação — 8k entidades.

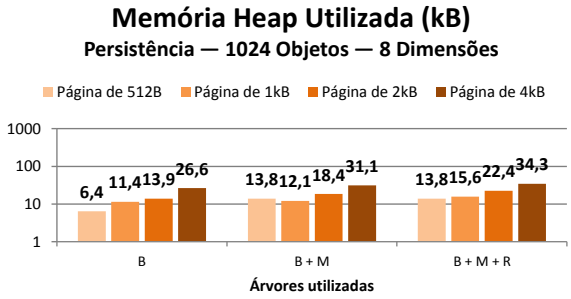


(g) Operação de Persistência — 16k entidades.

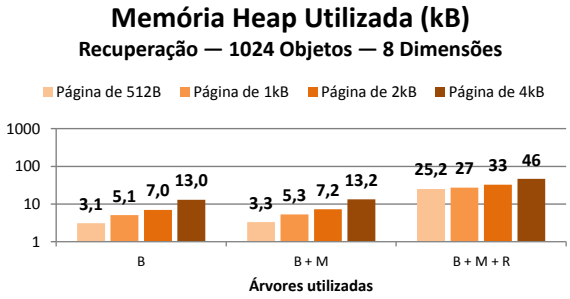


(h) Operação de Recuperação — 16k entidades.

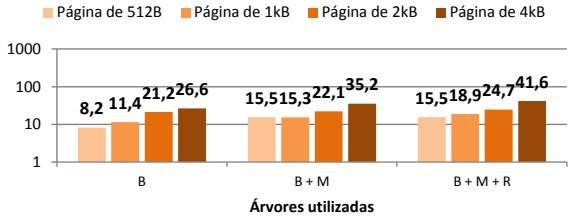
Figura 34: Resultados de consumo de memória *heap* obtidos para entidades de 4 dimensões em relação à quantidade de estruturas e o tamanho da página.



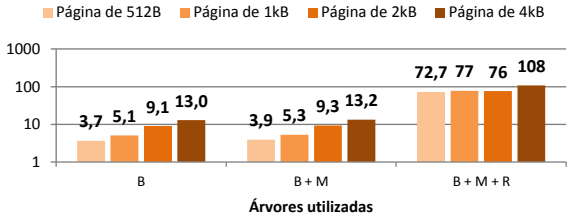
(a) Operação de Persistência — 1k entidades.



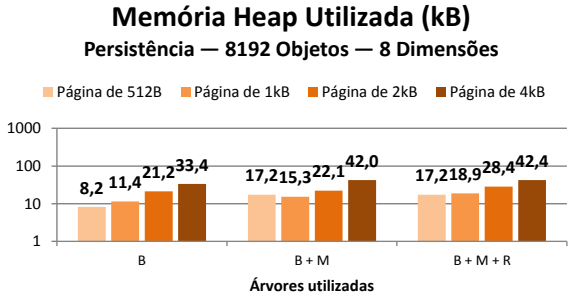
(b) Operação de Recuperação — 1k entidades.



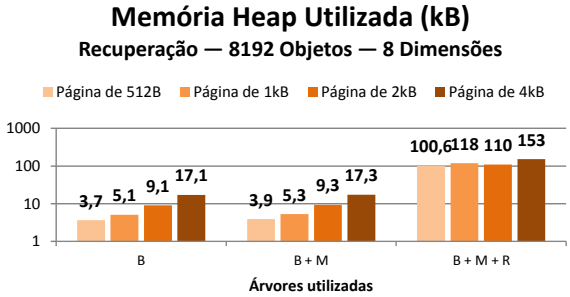
(c) Operação de Persistência — 4k entidades.



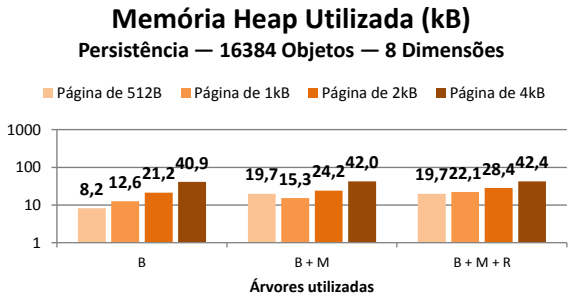
(d) Operação de Recuperação — 4k entidades.



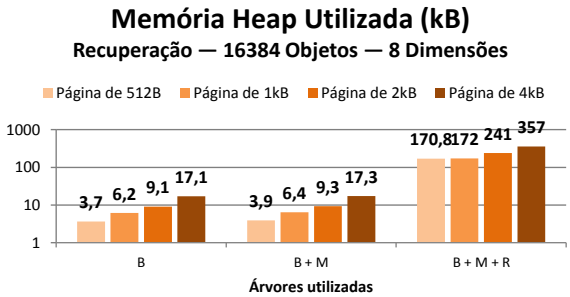
(e) Operação de Persistência — 8k entidades.



(f) Operação de Recuperação — 8k entidades.

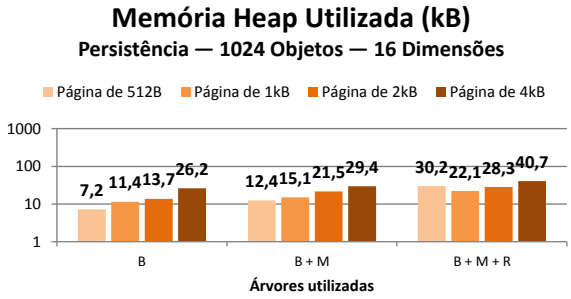


(g) Operação de Persistência — 16k entidades.

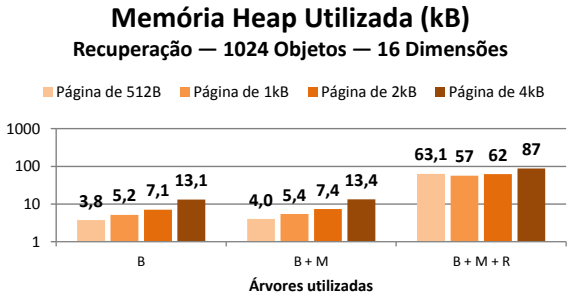


(h) Operação de Recuperação — 16k entidades.

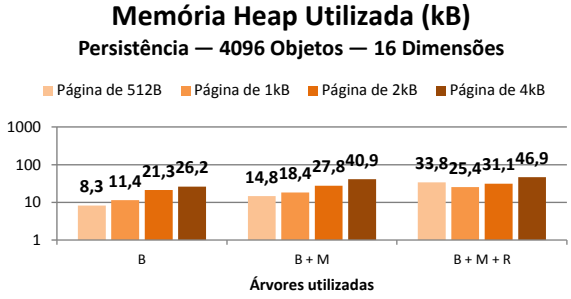
Figura 35: Resultados de consumo de memória *heap* obtidos para entidades de 8 dimensões em relação à quantidade de estruturas e o tamanho da página.



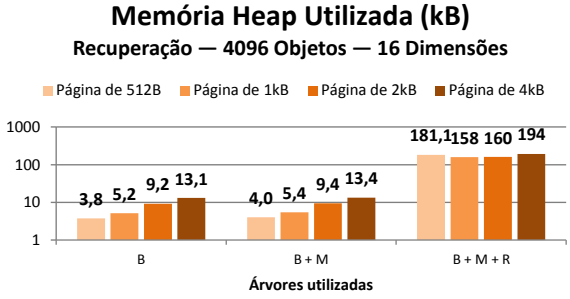
(a) Operação de Persistência — 1k entidades.



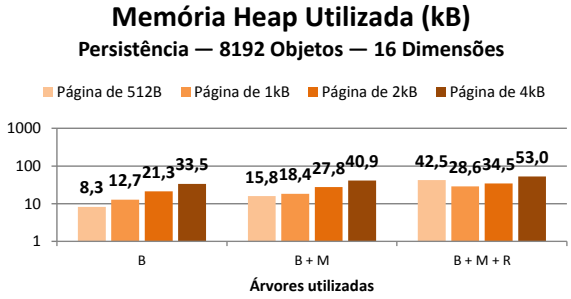
(b) Operação de Recuperação — 1k entidades.



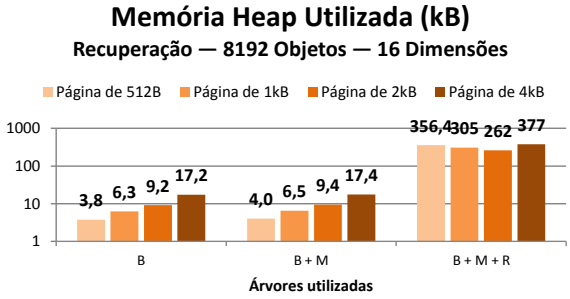
(c) Operação de Persistência — 4k entidades.



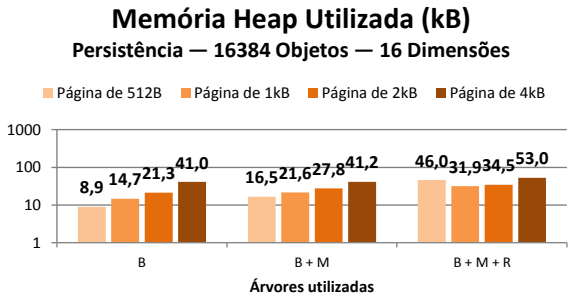
(d) Operação de Recuperação — 4k entidades.



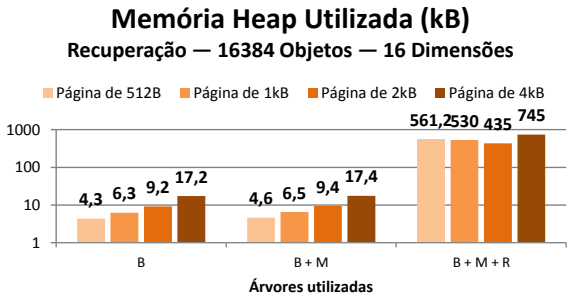
(e) Operação de Persistência — 8k entidades.



(f) Operação de Recuperação — 8k entidades.

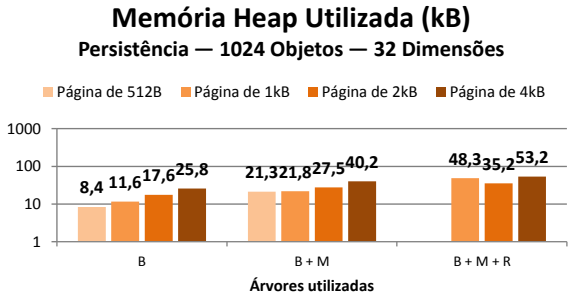


(g) Operação de Persistência — 16k entidades.

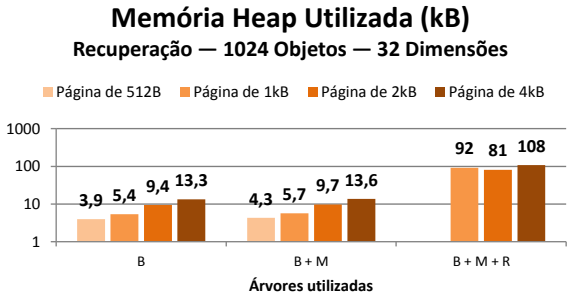


(h) Operação de Recuperação — 16k entidades.

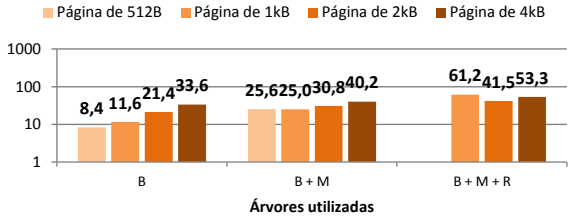
Figura 36: Resultados de consumo de memória *heap* obtidos para entidades de 16 dimensões em relação à quantidade de estruturas e o tamanho da página.



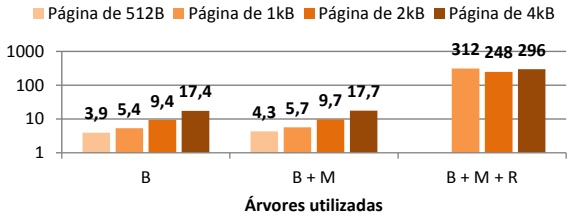
(a) Operação de Persistência — 1k entidades.



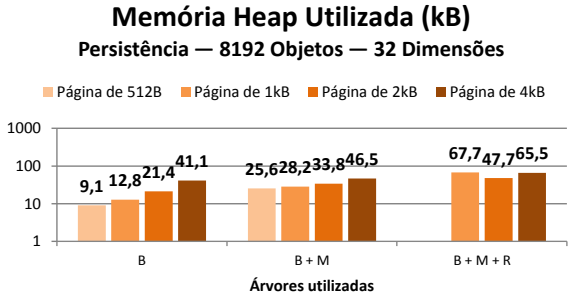
(b) Operação de Recuperação — 1k entidades.



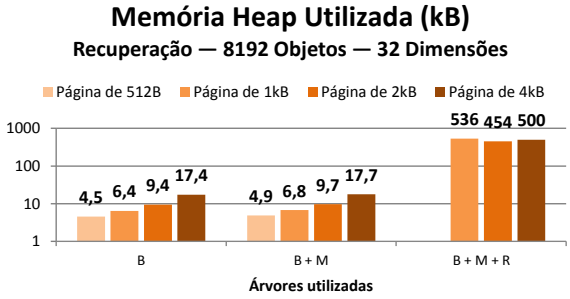
(c) Operação de Persistência — 4k entidades.



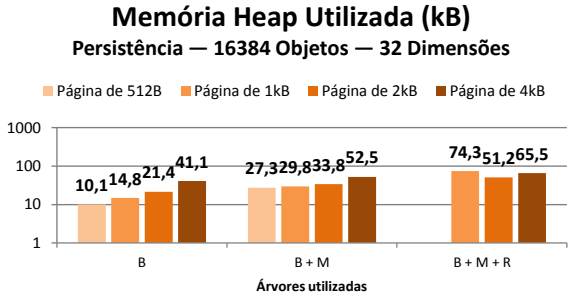
(d) Operação de Recuperação — 4k entidades.



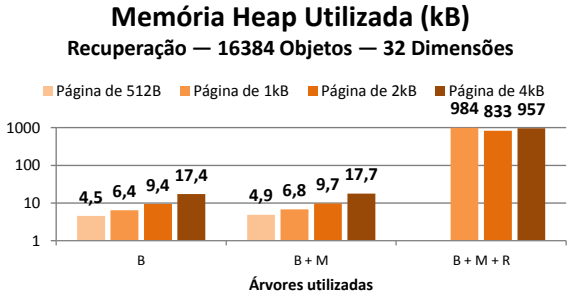
(e) Operação de Persistência — 8k entidades.



(f) Operação de Recuperação — 8k entidades.



(g) Operação de Persistência — 16k entidades.



(h) Operação de Recuperação — 16k entidades.

Figura 37: Resultados de consumo de memória *heap* obtidos para entidades de 32 dimensões em relação à quantidade de estruturas e o tamanho da página. Devido ao tamanho da informação armazenada na Árvore R, não foi executado o teste de página de arquivo de 512B a combinação Árvore B + Árvore M + Árvore R.

O aumento do tamanho da página do arquivo de persistência, ao mesmo tempo que pode aumentar o consumo de memória pelo carregamento de páginas maiores, também pode resultar em uma diminuição da memória utlizada se a árvore resultante tenha a sua altura diminuída e sua densidade (quantidade de objetos por folha) aumentada. Em uma Árvore B, toda operação de persistência ou recuperação necessita percorrer a altura h da árvore, carregando no mínimo h páginas de s *bytes* de tamanho. Se ocorre uma queda na altura h da árvore com o aumento do tamanho s da página, a quantidade de memória total utilizada $h * s$ pode diminuir. Já no caso das Árvores R e M, o consumo de memória também é influenciado pela verificação de sobreposições de seus nós, sendo necessário o carregamento de páginas adicionais para cada um de seus níveis — aumentando assim o número total de páginas carregadas e consequentemente o consumo de memória.

Neste experimento, houve a diminuição da memória utilizada após o aumento do tamanho da página do arquivo em alguns casos, como por exemplo:

1. Para entidades de 16 dimensões, no aumento de página de 512B para 1kB em ambas as operações de persistência e recuperação utilizando Árvores B, M e R;
2. Para entidades de 32 dimensões, no aumento de página de 1kB para 2kB em ambas as operações de persistência e recuperação utlizando Árvores B, M e R.

Outra característica de todos os casos testados é o grande impacto da utilização da Árvore R no consumo de memória, especialmente para operações de busca, atestando o alto custo do algoritmo utilizado. As Árvores B e M, entretanto, apresentaram uso de memória mais adequado para a execução de projetos em plataformas de menor custo, chegando a um pouco mais de 50kB no pior caso.

É importante ressaltar que, apesar de não ser possível supervisionar o uso da pilha de *stack* no ambiente simulado — e consequentemente identificar o estouro de *stack* —, as tarefas podem ser criadas utilizando o tamanho pretendido. O espaço será reservado na memória *heap* e levado em consideração na estimativa de consumo de memória. Assim, é possível obter através do projeto de simulação uma estimativa inicial da quantidade de memória necessária para a aplicação desejada.

4.3 Teste Funcional

O objetivo deste experimento foi comprovar o funcionamento da adaptação do *framework* Object-Injection para sistemas embarcados utilizando o sistema FreeRTOS™. Para tanto, foi criado um exemplo de aplicação com duas classes diferentes e relacio-

nadas, que foram então persistidas e depois recuperadas a partir de seus identificadores (UUID's) ou chaves.

O experimento foi realizado em duas etapas, sendo primeiramente executado em ambiente simulado no sistema operacional Windows (Seção 4.3.1) e posteriormente transferido para uma plataforma embarcada (Seção 4.3.2).

A base de dados deste experimento foi extraída do banco de dados público de inspeções sanitárias da cidade de Chicago (City of Chicago — Food Inspections, 2015). Os dados foram divididos em duas classes: `Facility`, que representa o estabelecimento onde foi feita a inspeção, e `Inspection`, que representa cada inspeção realizada. A modelagem do sistema e a relação entre as classes é mostrada pela Figura 38. A classe `InspectionDate` é auxiliar; seus dados são serializados juntamente com a classe `Inspection`.

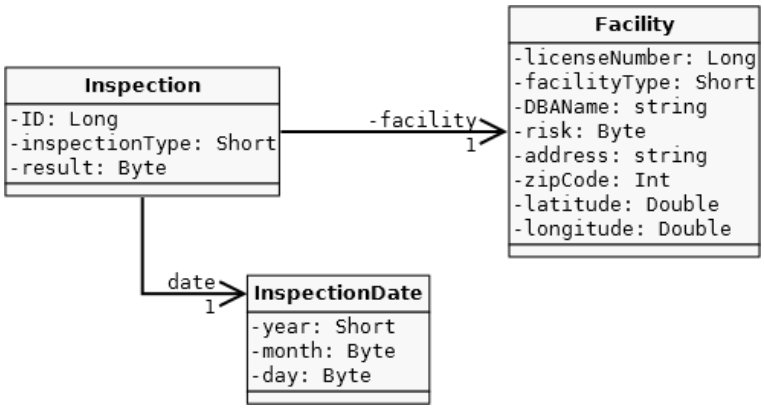


Figura 38: Classes `Facility` e `Inspection` utilizadas no Teste Funcional.

As classes foram implementadas juntamente com suas derivadas para persistência de entidades (classes `EntityFacility` e `EnitityInspection`) e indexação de chaves (classes `OrderFacility`, que indexa os estabelecimentos pelo número de sua licença (`licenseNumber`) em uma Árvore B, `PointFacility`, que indexa os estabelecimentos pela suas coordenadas — `latitude` e `longitude` — em uma Árvore M e `OrderInspection`, que indexa as inspeções pelo seu número de identificação (`ID`) em uma árvore B).

A interface da aplicação com o usuário é feita por meio de comandos implementados sobre a extensão CLI do FreeRTOS™. Para este experimento, foram criados os comandos:

- `BEGIN <page_size>:` cria/inicializa o arquivo de persistência com página de tamanho `<page_size>` e todas as estruturas de indexação das classes `Facility` e `Inspection`. Além disso, inicializa uma tarefa no sistema para controlar as operações referentes ao arquivo de persistência e duas filas (*queues*) para comunicação

entre a tarefa criada e a tarefa da extensão CLI.

- `END:` finaliza a aplicação, finalizando a tarefa e suas respectivas filas. Também fecha o arquivo de persistência e dealoca todas as estruturas de indexação.
- `ADD_FACILITY <facility_data>:` adiciona um registro da classe `Facility` ao arquivo de persistência. Os dados devem conter (1) o número da licença do estabelecimento (`license_number`), (2) o tipo de estabelecimento (`facility_type`), (3) o nome do estabelecimento (`name`), (4) o nível de risco (`risk`), (5) o endereço do estabelecimento (`adress`) e seu CEP (`zip_code`), (6) sua localização, através das suas coordenadas de latitude (`latitude`) e longitude (`longitude`) e (7) o UUID do registro. Os dados devem ser inseridos nesta ordem separados por um caracter de tabulação (`\t`).
- `ADD_INSPECTION <inspection_data>:` adiciona um registro da classe `Inspection` ao arquivo de persistência. Os dados devem conter (1) o identificador da inspeção (`ID`), (2) o ano da inspeção (`Year`), (3) o mês da inspeção (`Month`), (4) o dia da inspeção (`Day`), (5) o tipo da inspeção (`inspection_type`), (6) seu resultado (`result`), (7) o número da licença do local (`facility_license_number`) e (8) o UUID do registro. Os dados devem ser inseridos nesta ordem separados por um caracter de tabulação (`\t`).
- `FIND:` localiza registros de ambas as classes `Facility` e `Inspection` no arquivo de persistência. Os critérios de busca disponibilizados são:
  - `FIND INSPECTION UUID <uuid>:` recupera o registro de uma inspeção através do seu `UUID`. Neste caso, apenas uma operação de recuperação é realizada (na estrutura da classe `EntityInspection`).
  - `FIND INSPECTION ID <id>:` recupera o resgistro de uma inspeção através do seu `ID`. São realizadas duas operações: uma para a recuperação do `UUID` na estrutura da classe `OrderInspection`, e outra na da classe `EntityInspection`, utilizando o `UUID` encontrado para recuperar o registro.
  - `FIND FACILITY UUID <uuid>:` recupera o registro de um estabelecimento a partir do seu `UUID`. Apenas uma operação de recuperação é realizada (na estrutura da classe `EntityFacility`).
  - `FIND FACILITY LICENSE <license_number>:` recupera o registro de um estabelecimento através do número de sua licença. São feitas duas operações: a recuperação do `UUID` do registro na estrutura da classe `OrderFacility` e a recuperação do registro na estrutura da classe `EntityFacility`.
  - `FIND FACILITY COORD <latitude> <longitude>:` recupera o registro de um

estabelecimento através de suas coordenadas de latitude e longitude. São feitas duas operações: a recuperação do `UUID` do registro na estrutura da classe `PointFacility` seguida da recuperação do registro na estrutura da classe `EntityFacility`.

  - `FIND FACILITY INSPECTION UUID <uuid>:` recupera o registro do estabelecimento armazenado na inspeção identificada pelo `UUID` fornecido. Neste caso, são feitas três requisições. Primeiramente, o registro da inspeção é recuperado através do seu `UUID` da estrutura da classe `EntityInspection`; em seguida, o número de licença do estabelecimento do registro (`facility_license_number`) é utilizado para recuperar o `UUID` do estabelecimento através da estrutura da classe `OrderFacility`; por fim, o `UUID` é utilizado para recuperar o registro do estabelecimento da estrutura da classe `EntityFacility`.
  - `FIND FACILITY INSPECTION ID <id>:` recupera o registro do estabelecimento armazenado na inspeção identificada pelo `UUID` fornecido. Neste caso, são feitas quatro requisições. Primeiramente, o `UUID` da inspeção é recuperado através do seu `ID` da estrutura da classe `OrderInspection`; em seguida, o `UUID` é utilizado para recuperar o registro da inspeção da estrutura da classe `EntityInspection`; o número de licença (`facility_license_number`) do estabelecimento do registro recuperado é então utilizado para recuperar o `UUID` do estabelecimento através da estrutura da classe `OrderFacility`; por fim, o `UUID` é utilizado para recuperar o registro do estabelecimento da estrutura da classe `EntityFacility`.

- `HEIGHT:` recupera e imprime a altura de todas as estruturas utilizadas na demonstração.
- `HEAP:` utiliza os métodos disponíveis no sistema FreeRTOS™ para exibir os valores de memória *heap* livre e utilizados.

Os comandos que não acessam diretamente os blocos das estruturas de indexação (comandos `BEGIN`, `END`, `HEIGHT` e `HEAP`) foram implementados diretamente pela tarefa `CLI()`, que executa o processador de comandos. Para os demais, foi construída uma estrutura de comunicação entre a tarefa `CLI()` e uma tarefa que controla as operações de acesso ao arquivo de persistência (criada pelo comando `BEGIN` com o nome de `ObICtr()`) por meio de filas. Neste cenário, a tarefa `CLI()` faz uma requisição à `ObICtr()`, enviando o código da requisição e quaisquer dados necessários (por exemplo, no caso do comando `ADD_FACILITY`, é enviada a entidade serializada; no caso do comando `FIND`, é enviado o tipo de busca e o parâmetro utilizado). Uma vez feita a requisição, a tarefa `CLI()` é

suspensa até que uma resposta esteja disponível.

A tarefa `ObICtr()`, por sua vez, permanece em estado de suspensão até que receba uma requisição. Quando isso acontece, a tarefa realiza a ação requerida, e envia seu resultado e possíveis respostas (por exemplo, o valor de um `UUID` ou um registro serializado na execução do comando `FIND`) para a tarefa `CLI()`, voltando então ao estado de suspensão. A Figura 39 ilustra a troca de mensagens entre as duas tarefas quando um comando do tipo `FIND FACILITY INSPECTION ID <id>` é executado.

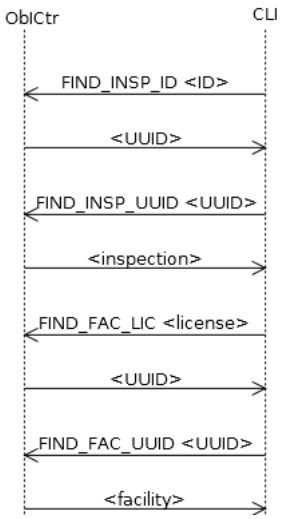


Figura 39: Troca de mensagens entre as tarefas `CLI()` e `ObICtr()` na execução do comando `FIND FACILITY INSPECTION ID <id>`.

A quebra de operações mais complexas é mais apropriada para um sistema preemptivo como o FreeRTOS™, que exige que nenhuma tarefa monopolize a execução, prejudicando seu comportamento de tempo real. O uso de filas e a suspensão das tarefas faz com que não sejam executados mais de um comando por vez e nem que o tempo de execução seja desperdiçado com tarefas que estejam "paradas" à espera de uma resposta (o que aconteceria se fosse utilizada uma técnica de *pooling*).

Com os comandos definidos, foram criados arquivos com todas as variações de comando criadas. Foram construídos arquivos para a persistência de todas as entidades da base de dados, além de arquivos de operações de recuperação utilizando todas as variações do comando `FIND`. Os arquivos de recuperação foram construídos de forma que todas as entidades fossem recuperadas pelo menos uma vez, seja pelo `UUID` ou por alguma das chaves.

4.3.1 Execução em Ambiente Simulado

Nesta primeira etapa de teste, a aplicação criada utiliza o FreeRTOS™ para o sistema operacional Windows utilizado nas outras atividades deste trabalho (Seção 3.5). Além do sistema FreeRTOS™, a aplicação inclui as extensões FAT SL (que implementa um disco FAT na memória RAM do computador) e CLI (que implementa uma interface de linha de comando através de um *socket* UDP).

Dado que a extensão FAT SL cria um disco FAT em memória RAM, foram criados dois comandos extras para que os arquivos de persistência criados pudessem ser transferidos para o disco rígido do computador:

- `DOWNLOAD <filename>:` realiza a cópia do arquivo especificado em `<filename>` do disco FAT virtual para o disco rígido do computador.
- `UPLOAD <filename>:` realiza a cópia do arquivo especificado em `<filename>` do disco rígido do computador para o disco virtual FAT.

A execução foi realizada em um máquina Intel® Core™ i7-3537U de 2.00GHz e 8,00GB de RAM executanto o sistema operacional Windows 8 Single Language. Os arquivos de comandos criados foram enviados para o *socket* de comunicação através do cliente YAT v1.99.34. Foram persistidas 14138 instâcias da classe `Facility` e 73516 instâncias da classe `Inspection` com sucesso. A execução dos comandos de busca gerados em arquivo não acusou nenhum erro, comprovando o funcionamento da adaptação do *framework* Object-Injection com o sistema FreeRTOS™.

4.3.2 Execução em Plataforma Embarcada

A execução desta segunda etapa foi realizada na plataforma de desenvolvimento para o microcontrolador LPC1769[4] em conjunto com a placa base compatível para prototipagem[5]. O LPC1769 é um ARM® Cortex-M3 com 64 kB de memória SRAM, 512 kB de memória *flash* e conexões UART, I2C, SPI, SSP, entre outras.

Foi desenvolvida uma aplicação que usa um *port* do sistema FreeRTOS™ desenvolvido para o microcontrolador NXP LPC1769[6] no ambiente de desenvolvimento LPCXpresso

[4] *LPC1769 LPCXpresso Board*
<https://www.embeddedartists.com/products/lpcxpresso/lpc1769_xpr.php>

[5] *LPCXPRESSO Base Board*
<https://www.embeddedartists.com/products/lpcxpresso/xpr_base.php>

[6] *FreeRTOS+ Featured Project for LPCXpress*
<http://interactive.freertos.org/entries/21334878-FreeRTOS-featured-project-for-LPCXpress>

(`https://www.lpcware.com/LPCXpresso`) que implementa a extensão FAT SL em um cartão SD através das conexões SPI ou SSP, e a extensão CLI através de uma porta UART, que pode ser acessada por um computador através do conversor USB–UART disponível na placa base.

A aplicação executa a versão 8.2.1 do sistema FreeRTOS™, e conta somente com os recursos mínimos de hardware para seu funcionamento (conexões UART e SSP). Como a memória SRAM do microcontrolador LPC1769 é divida em dois bancos não contíguos de 32kB cada, foi definida a macro `configAPPLICATION_ALLOCATED_HEAP` no arquivo `FreeRTOSConfig.h`, o que permitiu alocar o *array* `ucHeap` utilizado para a alocação dinâmica de memória do FreeRTOS™ no banco superior de memória SRAM.

A Figura 40 mostra o resultado da compilação da aplicação desenvolvida em conjunto com o sistema FreeRTOS™. É estimado (LPCXPRESSO..., 2015) um consumo de 231,3 kB de memória *flash* e aproximadamente 43kB de memória RAM.

```
   text    data     bss     dec     hex    filename
 203668   33212   10692  247572   3c714    FreeRTOS-Plus-IO-ObI-FATSL-CLI-Demo.axf
```

Figura 40: Resultado da compilação do conjunto Object-Injection + FreeRTOS™ na plataforma embarcada.

Realizada a transferência da aplicação, o teste foi executado na plataforma embarcada. Como na primeira etapa, os arquivos de comandos criados foram enviados através da conexão serial através do cliente YAT v1.99.34 e executados sem erros. O pico de utilização de memória *heap* neste teste foi de 31704 *bytes*. Não houve, entretanto, nenhuma operação que retornasse erro de alocação de memória ou estouro da pilha de *stack*.

A título de ilustração, uma pequena demonstração dos comandos implementados é mostrada a seguir. A Figura 41 exibe a inicialização da demonstração, cujas condições iniciais são obtidas através dos comandos `task-stats` (tarefas em execução), `heap` (uso de memória) e `dir` (arquivos presentes no cartão SD). O comando `BEGIN 512` inicializa a tarefa `ObICtr` e cria o arquivo de persistência `INSPECS.OBI` no cartão SD, com um tamanho de página de 512 *bytes*. A criação da tarefa é comprovada pela segunda execução do comando `task-stats`, e a criação do arquivo `INSPECS.OBI` é comprovada pela segunda execução do comando `dir`. O arquivo possui 3072 *bytes* de tamanho, ou seja, 6 páginas de 512 *bytes*. As páginas correspondem ao Bloco Cabeçalho e os Blocos Descritores de cada uma das cinco estruturas utilizadas nesta demonstração (Seção 2.3.5) — as quais

ainda não possuem nenhum dado persistido, e por isso têm altura zero, conforme exibido pelo do comando `height`.

```
task-stats
Task            State Priority Stack    #
*********************************************
UARTCmd           R     1       297     1
IDLE              R     0        70     2
Tmr Svc           B     4        58     3

heap
Total heap allocated: 32768
Heap at use: 16200
Free heap: 16568
Peak heap usage: 16304

dir
Error: f_findfirst() failed.

begin 512
SUCCESS
```

```
task-stats
Task            State Priority Stack    #
*********************************************
UARTCmd           R     1       210     1
IDLE              R     0        70     2
Tmr Svc           B     4        58     3
OBICtr            B     1       324     4

heap
Total heap allocated: 32768
Heap at use: 19544
Free heap: 13224
Peak heap usage: 19680

dir
INSPECS.OBI [writable file] [size=3072]

height
Facility BTreeEntity: 0
Facility BTree: 0
Facility MTree: 0
Inspection BTreeEntity: 0
Inspection BTree: 0
SUCCESS
```

Figura 41: Inicialização da demonstração. O comando `BEGIN 512` inicializa a tarefa `ObICtr` e cria o arquivo de persistência `INSPECS.OBI` no cartão SD, com um tamanho de página de 512 *bytes*. Como nenhuma operação de persistência foi realizada, a altura de todas as estruturas utilizadas é zero, conforme demonstrado pelo comando `height`.

A Figura 42 mostra o resultado após a persistência de quatro registros da classe `Facility` e 14 registros da classe `Inspection`. O tamanho do arquivo (7168 *bytes*) acusa a inclusão de 8 novas páginas de 512 *bytes* cada. Pela altura das estruturas, é possível perceber que as estruturas `FacilityBTreeEntity`, `FacilityBTree`, `FacilityMTree` e `InspectionBTree` receberam uma página cada uma, e as quatro páginas restantes fazem parte da estrutura `InspectionBTreeEntity`, cuja altura indica a ocorrência de operações de *split*.

A Figura 43 exibe o resultado de alguns comandos de recuperação. O comando `find inspection uuid` recupera a entidade a partir do valor do UUID, realizando apenas uma operação de recuperação. O comando `find facility coord` recupera a entidade a partir do valor de uma de suas chaves, realizando duas operações de recuperação (a primeira operação recupera o UUID da entidade e a segunda a entidade em si). Os comandos `find facility inspection uuid` e `find facility inspection id` recuperam uma entidade da classe `Facility` através do atributo `license_number` persistido na entidade da classe

```
heap
Total heap allocated: 32768
Heap at use: 19352
Free heap: 13416
Peak heap usage: 23392

dir
INSPECS.OBI [writable file] [size=7168]
```

```
height
Facility BTreeEntity: 1
Facility BTree: 1
Facility MTree: 1
Inspection BTreeEntity: 2
Inspection BTree: 1
SUCCESS
```

Figura 42: Resultados do sistema após persistência de entidades das classes `Facility` e `Inpection`.

```
find inspection uuid
    13ca566e-2c7c-3fc5-43c6-93a33598cf65
INSPECTION
ID: 509497
Date: 22-12-2011 (d-m-y)
Inspection Type: CANVASS (4)
Inspection Result: PASS (5)
Facility License Number: 1403725

find facility coord
    41.9172942445567003 -87.7065452894948976
FACILITY
License Number: 2428845
Facility Type: RESTAURANT (150)
Name: CEMITAS PUEBLA
Risk: HIGH (1)
Address: 3129 W ARMITAGE AVE
Zip Code: 60647
Latitude: 41.917294
Longitude: -87.706545
```

```
find facility inspection uuid
    2cbd3056-60d8-27ff-2ac8-485804f55272
FACILITY
License Number: 2437220
Facility Type: RESTAURANT (150)
Name: DUNKIN DONUTS/BASKIN ROBBINS
Risk: MEDIUM (3)
Address: 11525 S HALSTED ST
Zip Code: 60628
Latitude: 41.684250
Longitude: -87.641981

find facility inspection id 1442217
FACILITY
License Number: 2428845
Facility Type: RESTAURANT (150)
Name: CEMITAS PUEBLA
Risk: HIGH (1)
Address: 3129 W ARMITAGE AVE
Zip Code: 60647
Latitude: 41.917294
Longitude: -87.706545
```

Figura 43: Resultados de comandos de recuperação. São realizadas recuperações de entidades a partir do valor do seu UUID ou de uma de suas chaves indexadas. Também é possível recuperar entidades a partir de relações estabelecidas entre as classes da demonstração.

`Inspection`, que é recuperada pelo seu UUID ou pelo valor de sua chave primária (atributo `ID`). São realizadas, respectivamente, três e quatro operações de recuperação. Em todos os comandos, os registros são recuperados integralmente.

Por fim, a Figura 44 exibe o resultado do comando de finalização. O comando `END` ternima o uso do arquivo de persistência e exclui a tarefa `OBICtr` do escalonador do sistema, conforme mostrado pelo resultado do comando `task-stats`.

```
end
SUCCESS

task-stats
Task            State Priority Stack    #
************************************************
UARTCmd           R     1        72     1
IDLE              R     0        70     2
Tmr Svc           B     4        58     3
```

Figura 44: Finalização da demonstração. O comando `END` ternima o uso do arquivo de persistência e exclui a tarefa `OBICtr` do escalonador do sistema, conforme mostrado pelo resultado do comando `task-stats`.

4.4 Considerações Finais

Este Capítulo apresentou os resultados dos experimentos realizados para verificar a compatibilidade entre as implementações C++ e Java do *framework* Object-Injection, a portabilidade da implementação C++ e o funcionamento da adaptação do *framework* para o sistema FreeRTOS™.

O Teste de Compatibilidade verificou que a padronização dos tipos de dados e das operações de serialização e desserialização permite o compatilhamento correto de dados de forma transparente para o usuário. A padronização também contribui com a portabilidade da implementação C++ do *framework* — a utilização de variáveis de tamanho fixo garante o tipo de dados utilizados independentemente da arquitetura do processador ou compilador utilizados.

O Teste Estimativo fornece uma ferramenta aos desenvolvedores para realizar uma estimativa inicial de uso de memória *heap* utilizado pelo conjunto Object-Injection + FreeRTOS™ em um simulador no sistema operacional Windows. Os resultados mostraram que a utiliação das Árvores B e M apresentam utilização de memória dinâmica compatível com microcontroladores de menor custo.

Por fim, o Teste Funcional verificou a funcionalidade da adaptação em uma plataforma embarcada NXPResso com um , realizando a persistência e recuperação de objetos em Árvores B e M com sucesso. Foi utilizada uma plataforma de desenvolvimento para o microcontrolador LPC1769, um ARM® Cortex-M3 com 64 kB de memória SRAM e 512kB de memória *flash*.

5 Conclusão

O desenvolvimento de aplicações para sistemas embarcados que demandam armazenamento e recuperação de informações deve considerar problemas de interoperabilidade e portabilidade, uma vez que são construídas para diferentes plataformas e/ou linguagens de programação. Muitas vezes, existem grandes restrições de *hardware* que não permitem a utilização de grande parte dos SGBDs. Além disso, o armazenamento deve considerar a existência de novos tipos de dados, não-tradicionais, que surgem com o desenvolvimento de novas tecnologias, como por exemplo a *Internet das Coisas*.

A implementação na linguagem C++ e sua abordagem orientada a objetos tornam o *framework* Object-Injection um bom candidato para a construção de aplicações gerenciadoras de dados em sistemas embarcados, uma vez que fornecem fácil acesso ao *hardware* deste tipo de sistema e boa capacidade de customização.

Este trabalho apresentou a adaptação da implementação C++ do *framework* Object-Injection para operação sem o uso de um sistema operacional padrão, possibilitando sua aplicação em sistemas embarcados de baixo custo. Foi realizada a adaptação da implementação C++ do *framework* para o sistema FreeRTOS™, permitindo o desenvolvimento de aplicações de tempo real que executem múltiplas tarefas, além de potencializar o uso do *framework* no mercado dada a grande quantidade de plataformas de *hardware* que possuem uma adaptação para o sistema FreeRTOS™.

Para atingir o objetivo deste trabalho, foram estudadas questões relativas à implementação do *framework* em si e do sistema FreeRTOS™, além de aspectos de normatização das linguagens C e C++ e questões de portabilidade de *software* e tecnologias de armazenamento. O estudo resultou em modificações do *framework* em ambas as implementações C++ e Java para assegurar a compatibilidade entre elas e aumentar a portabilidade do Object-Injection, além de melhorar o desempenho do *software* pela redução do número de operações em disco.

Para o uso do *framework* Object-Injection em sistemas embarcados, é necessário for-

necer um novo dispositivo de armazenamento a partir da classe `AbstractWorkspace` para que seja realizada a persistência das entidades e chaves. No caso específico da adaptação para o sistema FreeRTOS™, foi criado um novo dispositivo utilizando a extensão FAT SL. Além disso, foram feitas adaptações para o uso de semáforos e do gerenciador de memória dinâmica.

Experimentos realizados comprovaram a compatibilidade entre as implementações do *framework*, que podem compartilhar dados de forma transparente, além da portabilidade da implementação C++, que foi facilmente transferida para a plataforma embarcada BeagleBone Black sem nenhuma modificação.

Também, foram realizados testes tanto em ambiente simulado em *desktop* quanto em plataforma embarcada real, nos quais as operações de persistência e recuperação de entidades e chaves foram realizadas com sucesso, comprovando o funcionamento da adaptação. O uso do *framework* Object-Injection com o FreeRTOS™ é feito de forma simples, bastando apenas a definição de uma macro (`FREERTOS_COMPATIBILITY`).

Por fim, o conjunto do *framework* Object-Injection com o sistema FreeRTOS™ pode ser considerado um forte candidato para gerenciamento de dados em plataformas embarcadas, uma vez que:

1. A construção do conjunto apresenta consumo de memória compatível com vários microcontroladores de **baixo custo** disponíveis no mercado;
2. A disponibilização de estruturas de **indexação métrica e espacial** (Árvores M e R) são um diferencial do *framework* em relação a outros protótipos disponíveis, que implementam aplicações relacionais, e que podem ser valiosas em aplicações de geoprocessamento e localização;
3. O sistema FreeRTOS™ fornece ferramentas para supervisão de pilha de *stack* e erros de alocação de memória que, aliados a outros recursos como um contador *watchdog* do microcontrolador, tornam o sistema **robusto** para recuperação de erros;
4. O *framework* Object-Injection não necessita de nenhuma tarefa de manutenção, sendo intrinsecamente **autônomo**;
5. A modelagem orientada a objetos do *framework* facilita a inclusão/exclusão de funcionalidades, deixando-o **modularizado**;
6. A padronização dos tipos de variáveis aumenta a portabilidade do *framework*, que pode ser considerado **multiplataforma**.

5.1 Contribuições

São listadas como contribuições deste trabalho:

- Equiparação de estruturas disponíveis nas implementações C++ e Java, contribuindo com a compatibildade entre as implementações;
- Padronização dos tipos de variáveis do *framework*, contribuindo tanto para a compatibilidade entre as implementações quanto para a portabilidade entre plataformas;
- Melhoria do desempenho do *framework* Object-Injection através da diminuição do número de acessos em disco;
- Melhoria da documentação da implementação C++ do *framework*;
- Criação de projetos de demonstração funcionais do conjunto Object-Injection para ambiente simulado do sistema operacional Windows e plataforma embarcada.

5.2 Trabalhos futuros

São listados como possíveis atividades relacionadas a este trabalho:

- Análise de desempenho do conjunto Object-Injection + FreeRTOS™ em microcontroladores de baixo custo;
- Estudo de usabilidade do conjunto Object-Injection + FreeRTOS™;
- Estudo comparativo entre o Object-Injection e um banco de dados relacional em sistemas embarcados (como por exemplo o SQLite);
- Adaptação de novo sistema de arquivos que possibilite o manuseio simultâneo de mais de um arquivo;
- Melhoria do sistema de semáforos para suporte a operações de transação em sistemas multiprocessados;
- Inserção de novas funcionalidades como sincronização e segurança de dados.

5.3 Dificuldades Encontradas

Nesta Seção, são listadas algumas dificuldades teóricas e de implementação encontradas no decorrer do trabalho, bem como as ações realizadas para superá-las.

Organização das estruturas do *framework* Object-Injection em disco

Um dos grandes desafios do trabalho foi compreender como as estruturas de inde-

xação do *framework* Object-Injection eram organizadas em arquivo, tópico geralmente não coberto em disciplinas de banco de dados ou em seu uso em situações práticas. Foi necessário verificar a documentação da implementação Java do *framework* para fazer um levantamento da organização dos metadados dos vários tipos de blocos existentes. Tal estudo resultou na melhora da documentação de código da aplicação C++ do *framework* e também na compreensão de como são realizadas, a nível de acesso ao disco, as operações de persistência e recuperação das estruturas, o que facilita uma futura implementação de novos índices primários e secundários no *framework* Object-Injection.

Organização dos dados persistidos dentro dos blocos do *framework* Object-Injection

Além de realizar o estudo de como as estruturas são organizadas em disco, foi necessário compreender como é feita a serialização das entidades e chaves nas diversas estruturas de indexação para que fosse possível solucionar o erro de compatibilidade descrido pela Seção 3.3.2. Somente após esse estudo foi possível analisar os arquivos textos gerados pelo *ObInject File Fumper* e detectar o erro de inconsistência das operações de serialização e desserialização de dados.

Configuração da plataforma BeagleBone Black

Embora existam construções dos sistemas *Linux* já compilados para a plataforma BeagleBone Black, a realização da compilação cruzada de aplicações não é um processo trivial, agravado pela inexperiência da aluna com o sistema operacional em linha de comando.

Existem duas formas principais de realizar a compilação cruzada de processadores: (1) o compilador da plataforma embarcada é instalado na plataforma *desktop* e o executável é transferido para a plataforma embarcada após a compilação; ou (2) a compilação é feita dentro da própria plataforma embarcada através da configuração de um servidor remoto de compilação. Neste trabalho, a aluna optou pela segunda opção, utilizando um servidor de compilação remoto na IDE NetBeans seguindo o guia disponível em `http://mechomaniac.com/BeagleboardDevelopmentWithNetbeans`. Para que a compilação cruzada seja bem sucedida com essa configuração, também foi necessário configurar o ambiente de compartilhamento de arquivos.

Dificuldades de implementação de *software*

Nem sempre os entraves ao desenvolvimento foram originados de erros de implementação do *framework* Object-Injection. São citados:

1. **Leitura de arquivos binários no Microsoft Visual Studio:** a leitura de arquivos binários no MVS resultou em erro nas primeiras tentativas de adaptação do *framework* para o sistema FreeRTOS™ devido ao fato de o compilador interpretar o caracter `^Z (0x1A)` como fim de arquivo, o que não ocorria com o compilador GCC. Foi necessário inserir a flag `ios::binary` nas operações de abertura de arquivo para que os dados fossem lidos corretamente;
2. **Adaptação do *framework* Object-Injection para utilização de arquivo único:** no início deste trabalho, a implementação C++ do Object-Injection utilizava um arquivo para cada estrutura de indexação de um mesmo tipo. Como a extensão FATSL do FreeRTOS™ suporta a manipulação de apenas um arquivo por vez, foi necessário modificar a estrutura de alguns blocos do *framework* em ambas as implementações C++ e Java para que mais de uma estrutura do mesmo tipo pudesse ser armazenada em um mesmo arquivo. Também aqui, a compreensão de como as estruturas são armazenadas em disco foi fundamental;
3. **Construção de projeto de demonstração na plataforma embarcada:** embora um projeto de demonstração do sistema FreeRTOS™ estivesse disponível para a plataforma embarcada utilizada no Teste Funcional (Seção 4.3.2), a compilação conjunta dos códigos C e C++ fez necessária a construção de um novo projeto em branco, no qual as bibliotecas de acesso ao *hardware* e os arquivos do sistema FreeRTOS™ foram adicionados passo a passo, até a integração total do *framework* ao projeto. O processo foi feito por tentativa e erro e teve que ser repetido várias vezes até que fosse construído com sucesso;
4. **Leitura de comandos na plataforma embarcada:** na realização do teste funcional, foi observado que muitos comandos enviados pela porta serial não eram executados. Após várias verificações no ambiente de depuração, foram necessárias duas modificações: (1) aumento do tamanho do *buffer* de entrada para que comandos maiores fossem aceitos (o padrão do projeto era de 50 caracteres e foi aumentado para 200) e (2) o tipo da variável que realizava a navegação (*index*) do *buffer* de entrada teve que ser modificado para `unsigned char`, uma vez que originalmente, no tipo `char` padrão, só eram corretamente processados 127 caracteres, mesmo que o tamanho do *buffer* ultrapassasse esse valor.

APÊNDICE A – Programas Utilizados para Automatização de Testes

A.1 Testes de Escrita em Disco

Programa 19: *Script* utilizado para automatização dos testes de escrita em disco (Seção 3.4).

```
#!/bin/bash
#echo Initialize variables
pageSizes=(1024 2048 4096)
numObjects=(1 10 100 1000 10000 100000)
#echo for loop constructions
for size in ${pageSizes[@]}
do
   #echo Execute insertion
   for items in ${numObjects[@]}
   do
      for i in {1..30}
      do
          rm *.btree
          time ./student_add_"$size"_"$items" >> add_"$size"_"$items"_"$i".csv
#          echo "Movendo arvores ($items objetos)"
          cp  EntityStudent.btree Arvores/Entity_"$size"_"$items"_"$i".btree
          cp  PrimaryKeyStudent.btree Arvores/Keys_"$size"_"$items"_"$i".btree
      done
   done
#   echo "Execute queries"
   for items in ${numObjects[@]}
   do
       for i in {1..30}
       do
           time ./student_find_"$size"_"$items" >> find_"$size"_"$items"_"$i".csv
       done
   done
   mv EntityStudent.btree Arvores/Entity_"$size"_"$items"_"$i".btree
   mv PrimaryKeyStudent.btree Arvores/Keys_"$size"_"$items"_"$i".btree
done
#echo End of tests!
```

A.2 Teste de Compatibilidade

Programa 20: *Script* utilizado para extração dos dados.

```
clData <- function(){
## Read File
data <- read.csv("Crimes2014.csv")
## Select columns
data <- data[,c(1,2,4,5,8,9,14,16,17,20,21)]
## Remove missing data
data <- data[complete.cases(data),]
## Remove empty strings and NAs
for(i in colnames(data)){
        data<-(data[!(is.na(data[[i]]) | data[[i]]==""),])
}
## Remove zero-values
## ID
data <- data[data$ID > 0,]
## Community Area
data <- data[data$Community.Area > 0,]
## X Coordinate
data <- data[data$X.Coordinate > 0,]
## Y Coordinate
data <- data[data$Y.Coordinate > 0,]
## Latitude
data <- data[!(data$Latitude == 0),]
## Longitude
data <- data[!(data$Longitude == 0),]
nrow(data)

## Remove duplicates
## ID
data <- data[!duplicated(data$ID),]
## Case number
data <- data[!duplicated(data$Case.Number),]
## X and Y coordinates
data <- data[!duplicated(data[,c(8,9)]),]
## Latitude and longitude
data <- data[!duplicated(data[,c(10,11)]),]
nrow(data)

## Get 145k data
data <- data[1:148480,]
## Write to file
write.table(data,"Crimes2014ND.asc",quote=FALSE,row.names=FALSE,col.names=FALSE,sep="\t")
}
```

Programa 21: *Script* utilizado para execução das repetições da primeira etapa do teste (Seção 4.1).

```
@echo off
for %%P in (1024, 2048, 4096) do ( ::PageSize
    for /l %%R in (1,1,30) do (
        CrimeRecord_CPP_Persistencia_%%P.exe
        echo.
        CrimeRecord_CPP_Recuperacao_%%P.exe
        echo.
        java -jar CrimeRecord_JAV_Recuperacao_%%P.jar
        echo.
        move CrimeRecord.obi trees\CrimeRecord_CPP_%%P_%%R.obi > nul
    )
    for /l %%R in (1,1,30) do (
        java -jar CrimeRecord_JAV_Persistencia_%%P.jar
        echo.
        java -jar CrimeRecord_JAV_Recuperacao_%%P.jar
        echo.
        CrimeRecord_CPP_Recuperacao_%%P.exe
        echo.
        move CrimeRecord.obi trees\CrimeRecord_JAV_%%P_%%R.obi > nul
    )
)
```

Programa 22: *Script* utilizado para execução das repetições do primeiro passo da segunda etapa do teste (Seção 4.1).

```
#!/bin/bash
#echo Initialize variables
pageSizes=(1024 2048 4096)
#echo Create folder
for size in ${pageSizes[@]}
do
    for i in {1..30}
    do
        ./CrimeRecord_BBB_Persistencia_"$size"
  echo
        ./CrimeRecord_BBB_Recuperacao_"$size"
  echo
        mv CrimeRecord.obi /media/SD/Arvores/CrimeRecord_BBB_"$size"_"$i".obi
    done
done
```

Programa 23: *Script* utilizado para execução das repetições (ambiente *desktop*) — Segundo passo da segunda etapa do teste (Seção 4.1).

```
@echo off
for %%P in (1024, 2048, 4096) do ( ::PageSize
    for /l %%R in (1,1,30) do (
        copy trees\CrimeRecord_BBB_%%P_%%R.obi CrimeRecord.obi > nul
        CrimeRecord_CPP_Recuperacao_%%P.exe
        echo.
        del CrimeRecord.obi
    )
    for /l %%R in (1,1,30) do (
        copy trees\CrimeRecord_BBB_%%P_%%R.obi CrimeRecord.obi > nul
        java -jar CrimeRecord_JAV_Recuperacao_%%P.jar
        echo.
        del CrimeRecord.obi
    )
)
```

Programa 24: *Script* utilizado para execução das repetições (plataforma embarcada) — Segundo passo da segunda etapa do teste (Seção 4.1).

```
#!/bin/bash
#echo Initialize variables
pageSizes=(1024 2048 4096)

for size in ${pageSizes[@]}
do
    for i in {1..30}
    do
        cp /media/SD/CrimeRecord_CPP_"$size"_"$i".obi CrimeRecord.obi
        ./CrimeRecord_BBB_Recuperacao_"$size"
  echo
        rm CrimeRecord.obi
    done
    for i in {1..30}
    do
        cp /media/SD/Arvores/CrimeRecord_JAV_"$size"_"$i".obi CrimeRecord.obi
        ./CrimeRecord_BBB_Recuperacao_"$size"
        rm CrimeRecord.obi
    done
done
```

A.3 Teste Estimativo

Programa 25: *Script* utilizado para automatização do teste estimativo (Seção 4.2).

```
@echo off
for %%D in (04, 08, 16, 32) do ( :: Dimensions
    for %%P in (512, 1024, 2048, 4096) do ( ::PageSize
        for %%O in (1024, 4096, 8192, 16384) do ( ::NumberOfObjetcs
```

```
            for /l %%R in (1,1,30) do (
                ImageHistogram_B_Persistencia_%%D_%%P_%%O.exe
                copy ImgHist.obi trees\ImageHistogram_B_%%D_%%P_%%O_%%R.obi > nul
                if %%R NEQ 30 (
                    del ImgHist.obi > nul
                )
            )
            for /l %%R in (1,1,30) do (
                ImageHistogram_B_Recuperacao_%%D_%%P_%%O.exe
            )
            del ImgHist.obi > nul
        )
    )
)
for %%D in (04, 08, 16, 32) do ( :: Dimensions
    for %%P in (512, 1024, 2048, 4096) do ( ::PageSize
        for %%O in (1024, 4096, 8192, 16384) do ( ::NumberOfObjetcs
            for /l %%R in (1,1,30) do (
                ImageHistogram_BM_Persistencia_%%D_%%P_%%O.exe
                copy ImgHist.obi trees\ImageHistogram_BM_%%D_%%P_%%O_%%R.obi > nul
                if %%R NEQ 30 (
                    del ImgHist.obi > nul
                )
            )
            for /l %%R in (1,1,30) do (
                ImageHistogram_BM_Recuperacao_%%D_%%P_%%O.exe
            )
            del ImgHist.obi > nul
        )
    )
)
for %%D in (04, 08, 16, 32) do ( :: Dimensions
    for %%P in (512, 1024, 2048, 4096) do ( ::PageSize
        for %%O in (1024, 4096, 8192, 16384) do ( ::NumberOfObjetcs
            for /l %%R in (1,1,30) do (
                ImageHistogram_BMR_Persistencia_%%D_%%P_%%O.exe
                copy ImgHist.obi trees\ImageHistogram_BMR_%%D_%%P_%%O_%%R.obi > nul
                if %%R NEQ 30 (
                    del ImgHist.obi > nul
                )
            )
            for /l %%R in (1,1,30) do (
                ImageHistogram_BMR_Recuperacao_%%D_%%P_%%O.exe
            )
            del ImgHist.obi > nul
        )
    )
)
```

APÊNDICE B – Resultados Obtidos: Operações de Escrita em Disco

Este Apêndice apresenta integralmente os resultados obtidos nos testes de avaliação das operações de escrita em disco descritos pela Seção 3.4. Os testes foram executados em ambiente virtual executando o sistema operacional Ubuntu 12.04 e utilizando 4GB de memória RAM e dois núcleos da máquina hospedeira, de processador Intel® Core™ i7-3537U de 2.00GHz e 8,00GB de RAM executanto o sistema operacional Windows 8 Single Language.

O teste foi realizado em duas etapas: *persistência* e *recuperação*. Na primeira, entidades e chaves foram persistidas em arquivos distintos, que posteriormente foram utilizados para avaliações de operações de recuperação. Foram realizadas trinta (30) replicações de cada caso de teste (Apêndice A.1) para descartar influências externas nos resultados, sendo avaliados o tempo de execução (através do comando `time` do sistema operacional Ubuntu) e a frequência de escrita dos blocos em disco, ambos para diferentes tamanhos de página de arquivo (1kB, 2kB e 4kB).

Os resultados de tempo de execução são apresentados na forma de *boxplot*, enquanto que os resultados de operações de escrita em disco são apresentados em gráficos de barras categorizados conforme abaixo:

- **Bloco Cabeçalho** — número de vezes que o Bloco Cabeçalho foi escrito em disco;
- **Bloco Descritor** — número de vezes que o Bloco Descritor foi escrito em disco;
- **Outros Blocos** — soma do número de vezes que todos os blocos que não o Cabeçalho ou Descritor foram escritos em disco;
- **Pico de Escrita** — número de vezes que um bloco que não o Cabeçalho ou Descritor com a maior quantidade de atualizações foi escrito em disco.

B.1 Avaliação Inicial

Os resultados obtidos na avaliação inicial mostram que o tempo de execução do software diminui conforme o tamanho da página aumenta (Figura 45). Páginas maiores comportam uma maior quantidade de objetos, fazendo com que o número de operações de escrita em disco seja menor para a persistência de entidades (Figura 46) e chaves (Figura 47), tornando a execução mais rápida. Uma característica comum a todos os testes é a atualização em disco dos blocos mesmo na operação de recuperação, onde não há modificação das estruturas de indexação.

B.2 Condicionamento da Atualização em Disco

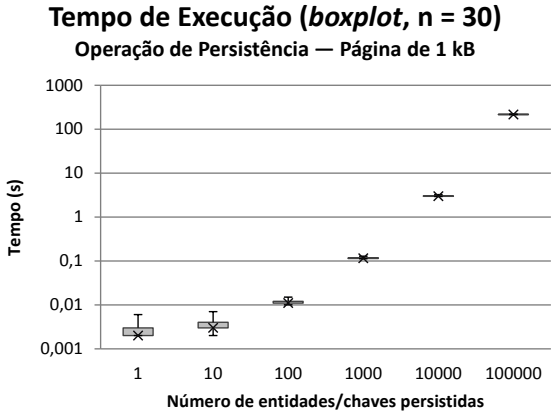
Os resultados obtidos após condicionamento da atualização dos dados em disco mostram diminuição significativa do tempo de execução, especialmente nas operações de recuperação, conforme mostrado pela Figura 48. A diferença entre os tempos de execução é explicada pela queda expressiva do número de atualizações em disco, tanto para a persistência de entidades (Figura 49) quanto para a de chaves (Figura 50).

Nas operações de persistência, em todos os casos, ocorre queda significativa do número de operações de escrita em disco, chegando a 60% para a categoria *Outros Blocos*. Nesta nova implementação, apenas o bloco folha é atualizado em uma persistência sem operações de *split*; o Bloco Descritor, é atualizado somente quando a altura da árvore muda.

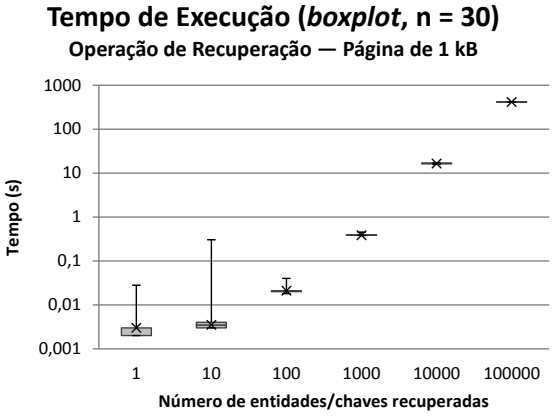
Nas operações de recuperação, os blocos da estrutura de indexação não são atualizados em disco, já que não são modificados. O bloco Cabeçalho continua sendo atualizado devido à atualização do seu registro *Last Session ID*, incrementado toda vez que o `workspace` é acessado.

B.3 Condicionamento da Atualização do Registro Last Session ID

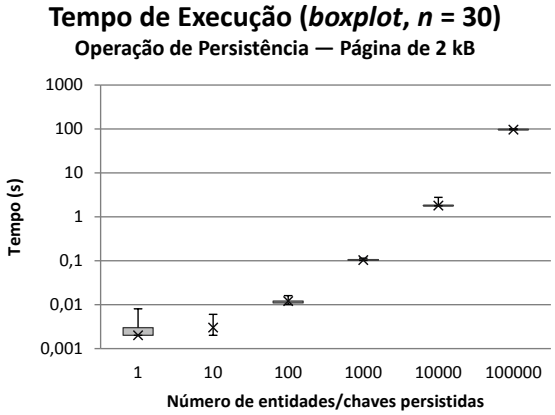
Os resultados obtidos sem a atualização do campo *Last Session ID* mostram uma pequena diminuição no tempo de execução para ambas as operações de persistência e recuperação, conforme exibido pela Figura 51, devido à diminuição do número de escritas do bloco Cabeçalho dos arquivos de persistência de entidades (Figura 52) e chaves (Figura 53).



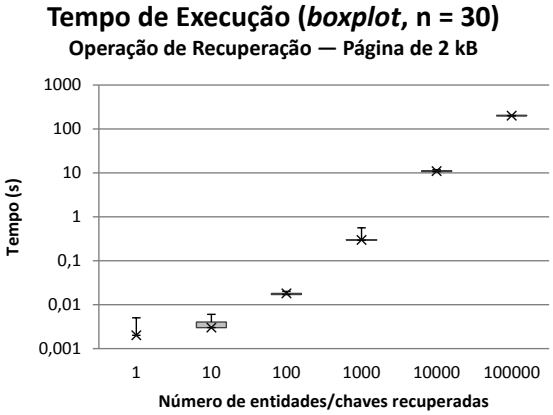
(a) Operação de Persistência — Página de 1kB



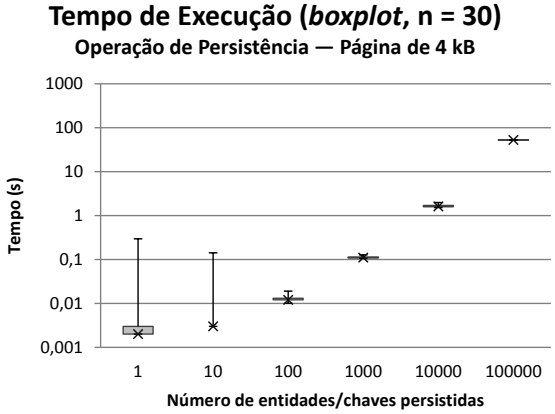
(b) Operação de Recuperação — Página de 1kB



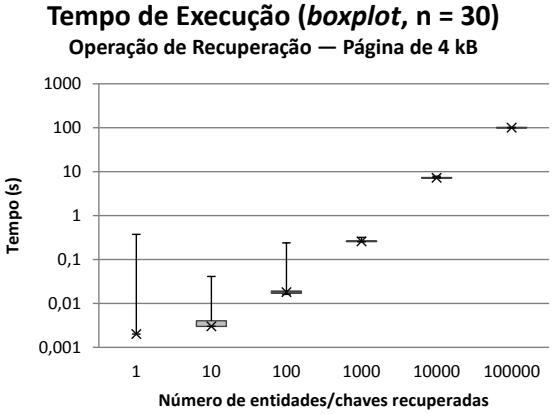
(c) Operação de Persistência — Página de 2kB



(d) Operação de Recuperação — Página de 2kB

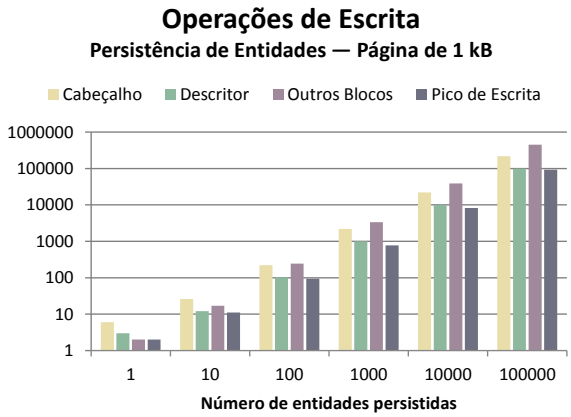


(e) Operação de Persistência — Página de 4kB

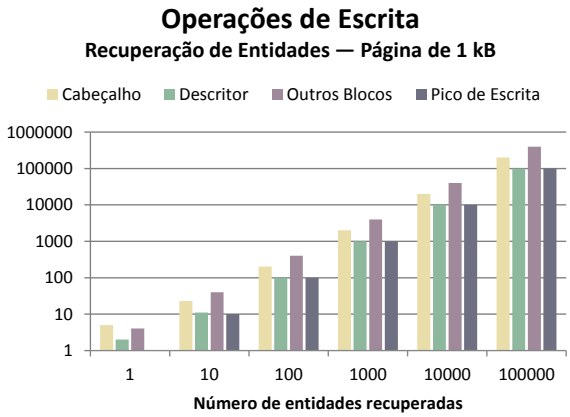


(f) Operação de Recuperação — Página de 4kB

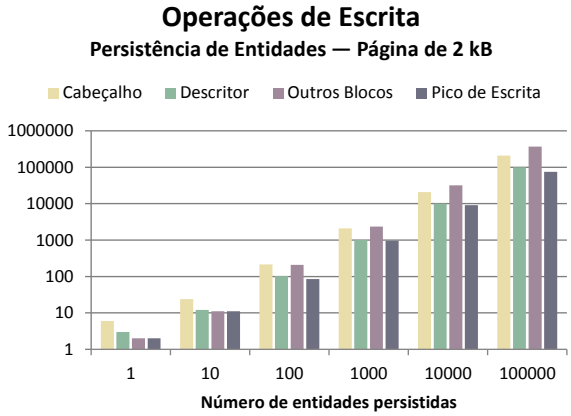
Figura 45: Avaliação Inicial: Tempo de execução.



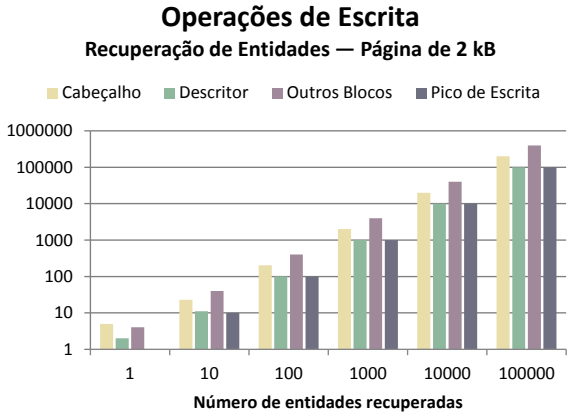
(a) Persistência de Entidades — Página de 1kB



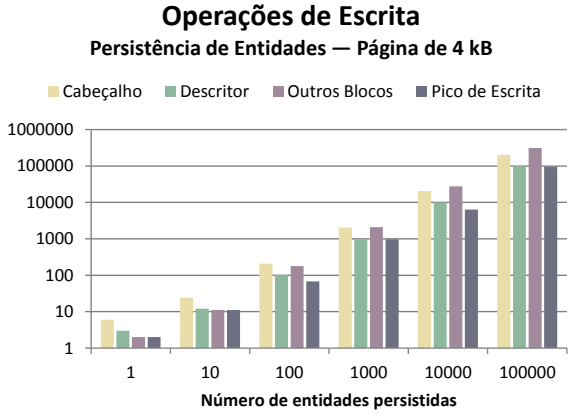
(b) Recuperação de Entidades — Página de 1kB



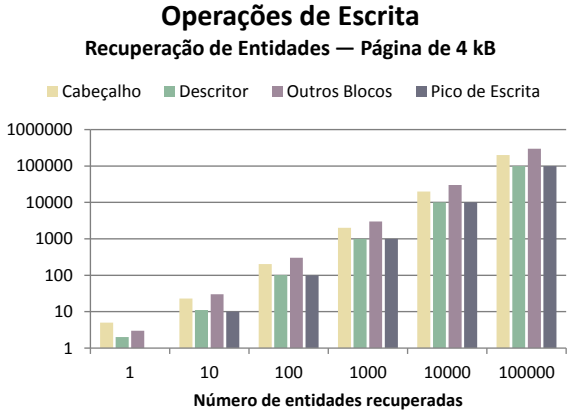
(c) Persistência de Entidades — Página de 2kB



(d) Recuperação de Entidades — Página de 2kB

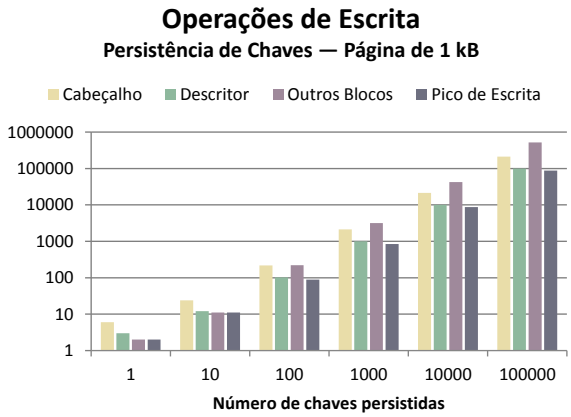


(e) Persistência de Entidades — Página de 4kB

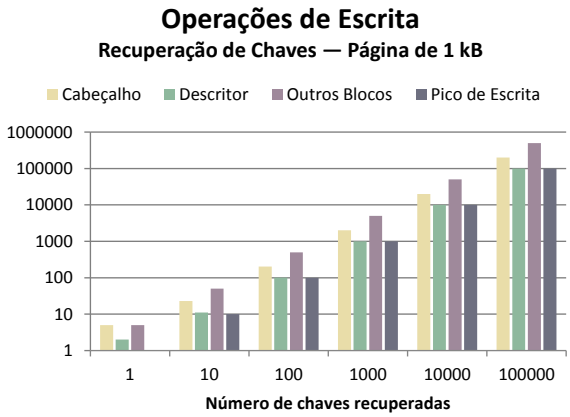


(f) Recuperação de Entidades — Página de 4kB

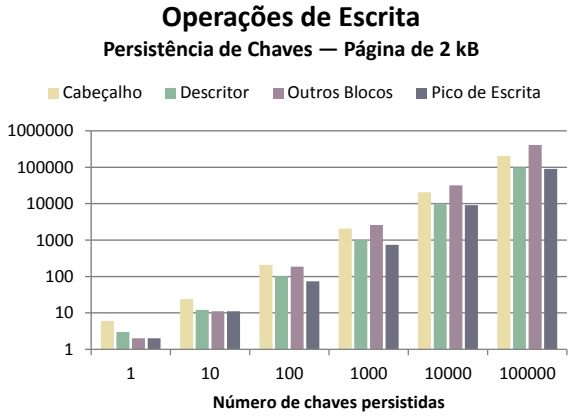
Figura 46: Avaliação Inicial: Número de operações de escrita em disco para persistência e recuperação de entidades.



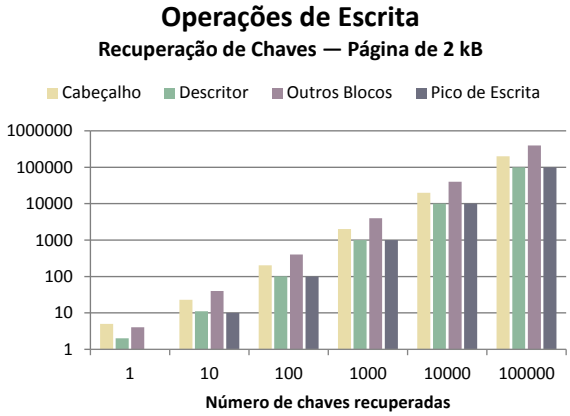
(a) Persistência de Chaves — Página de 1kB



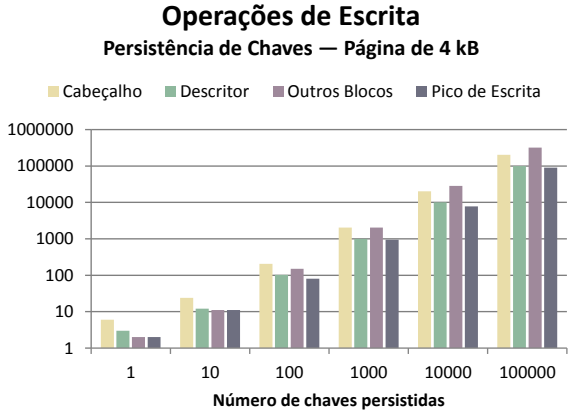
(b) Recuperação de Chaves — Página de 1kB



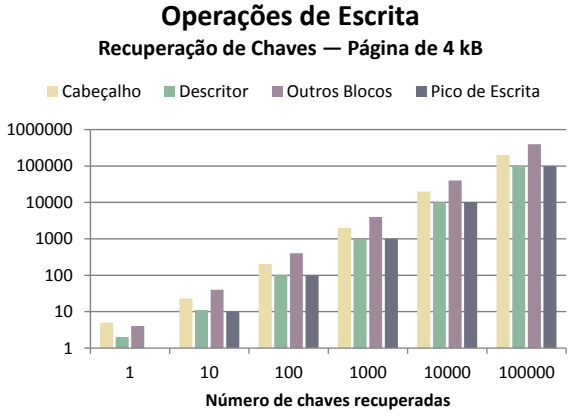
(c) Persistência de Chaves — Página de 2kB



(d) Recuperação de Chaves — Página de 2kB

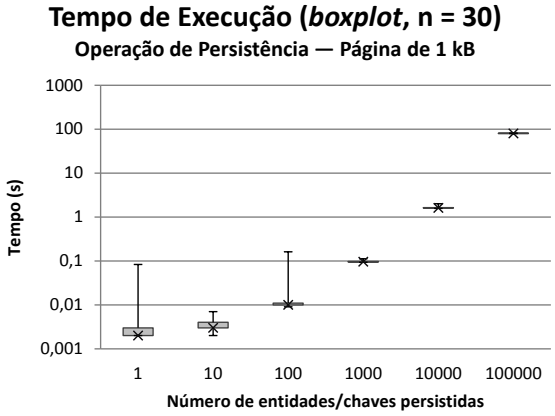


(e) Persistência de Chaves — Página de 4kB

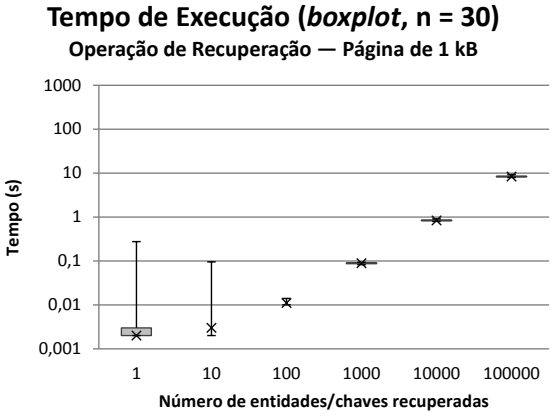


(f) Recuperação de Chaves — Página de 4kB

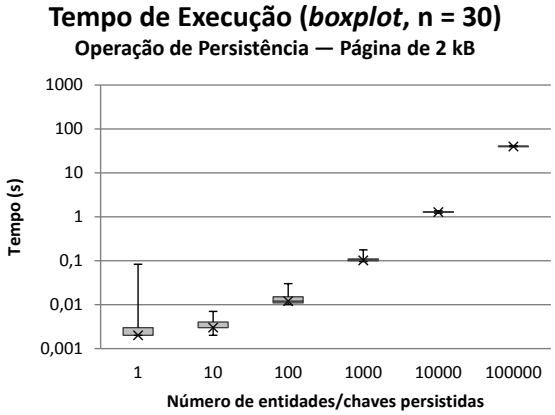
Figura 47: Avaliação Inicial: Número de operações de escrita em disco para persistência e recuperação de chaves.



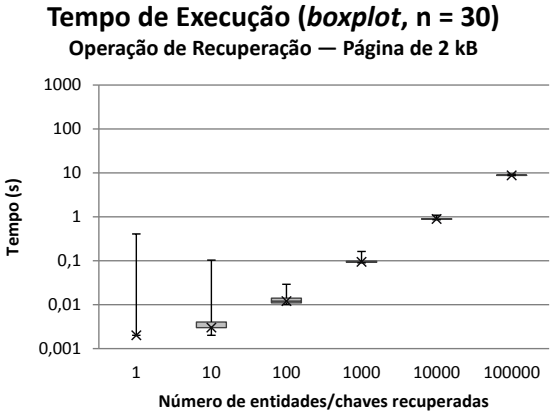
(a) Operação de Persistência — Página de 1kB



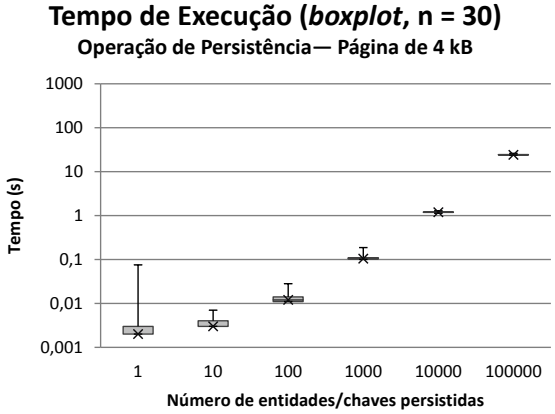
(b) Operação de Recuperação — Página de 1kB



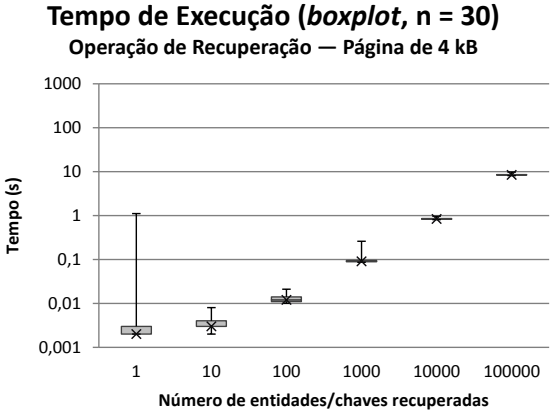
(c) Operação de Persistência — Página de 2kB



(d) Operação de Recuperação — Página de 2kB

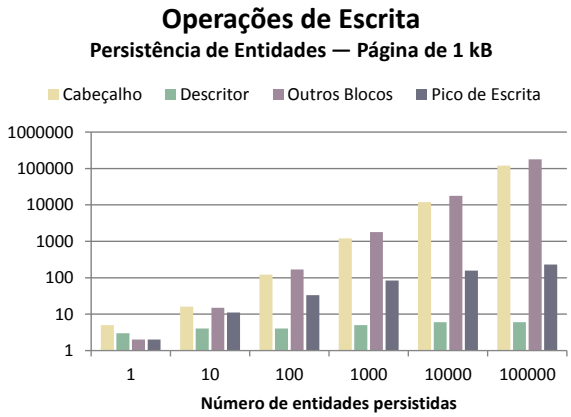


(e) Operação de Persistência — Página de 4kB

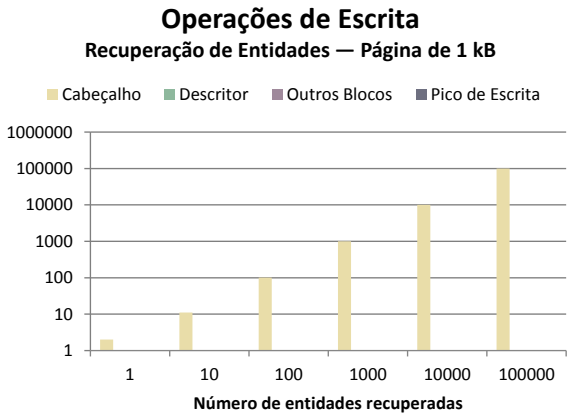


(f) Operação de Recuperação — Página de 4kB

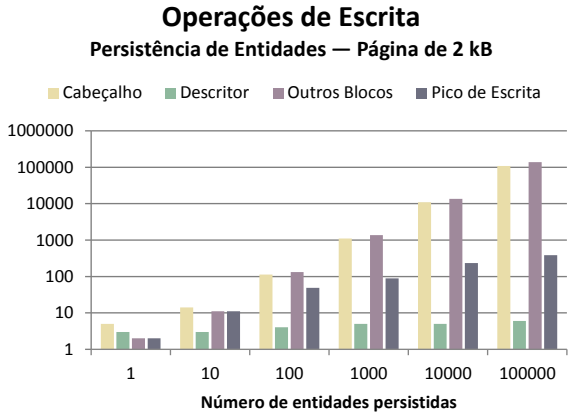
Figura 48: Condicionamento da Atualização em Disco: Tempo de execução.



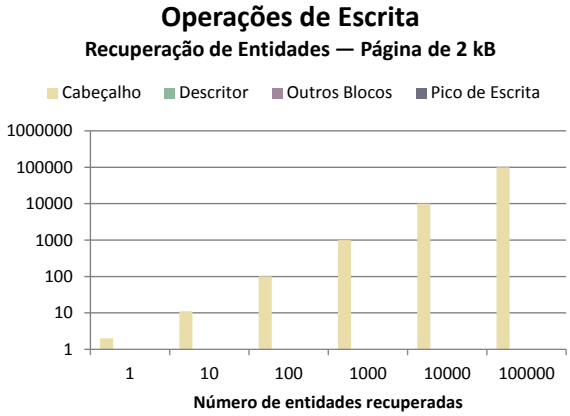
(a) Persistência de Entidades — Página de 1kB



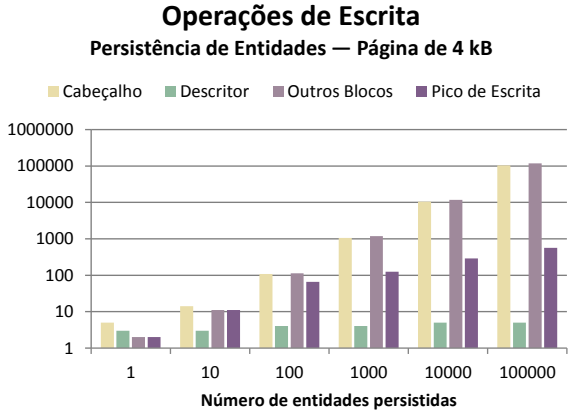
(b) Recuperação de Entidades — Página de 1kB



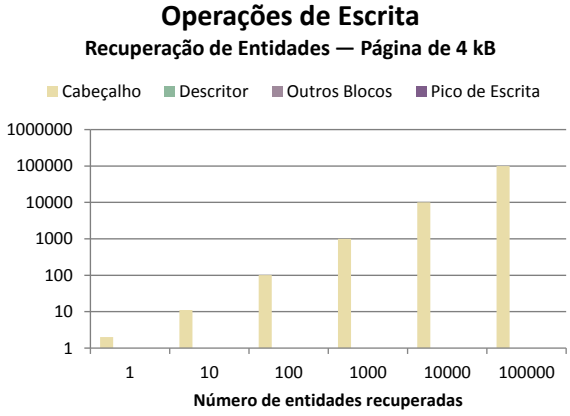
(c) Persistência de Entidades — Página de 2kB



(d) Recuperação de Entidades — Página de 2kB

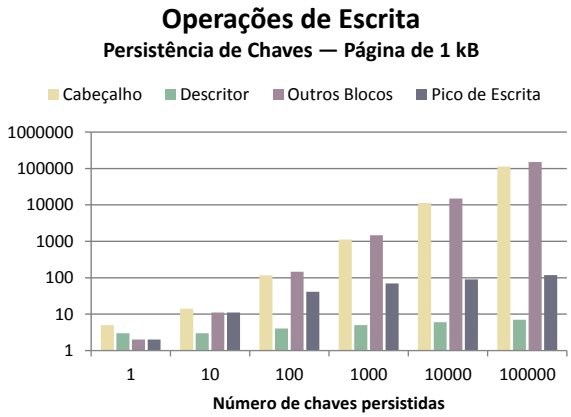


(e) Persistência de Entidades — Página de 4kB

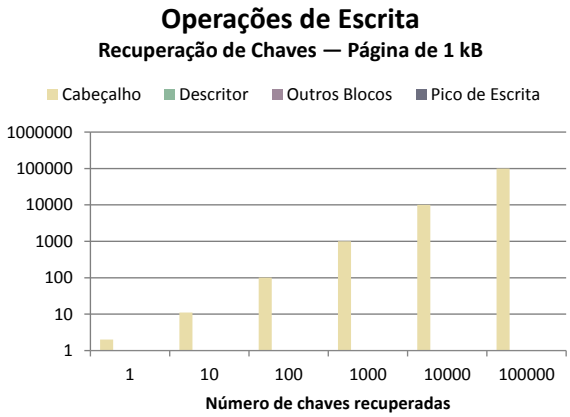


(f) Recuperação de Entidades — Página de 4kB

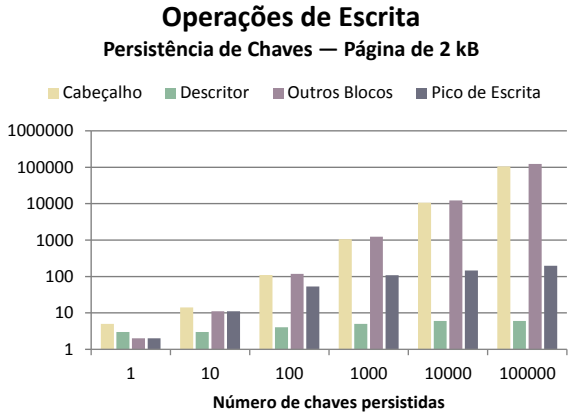
Figura 49: Condicionamento da Atualização em Disco: Número de operações de escrita em disco para persistência e recuperação de entidades.



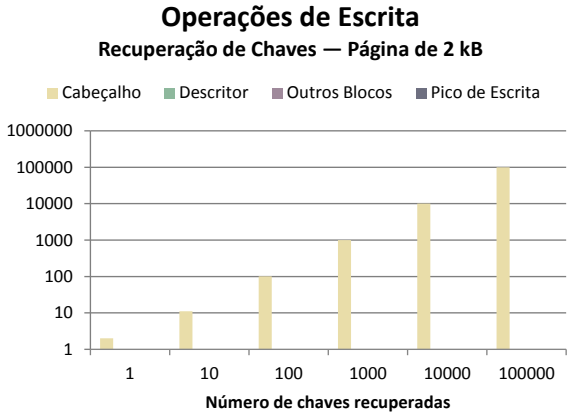
(a) Persistência de Chaves — Página de 1kB



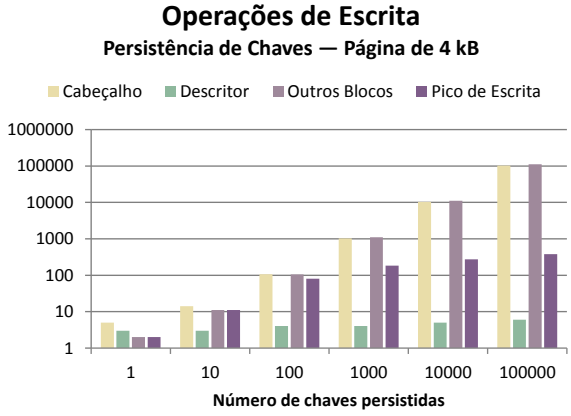
(b) Recuperação de Chaves — Página de 1kB



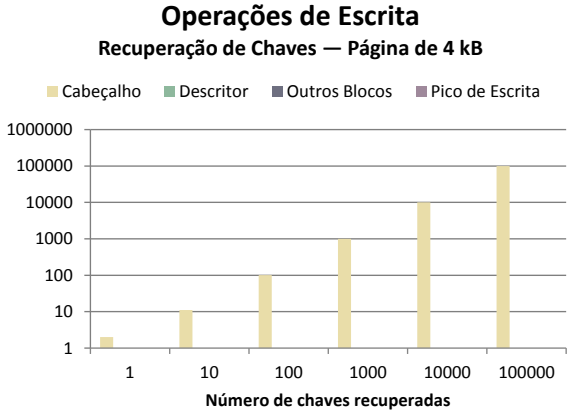
(c) Persistência de Chaves — Página de 2kB



(d) Recuperação de Chaves — Página de 2kB

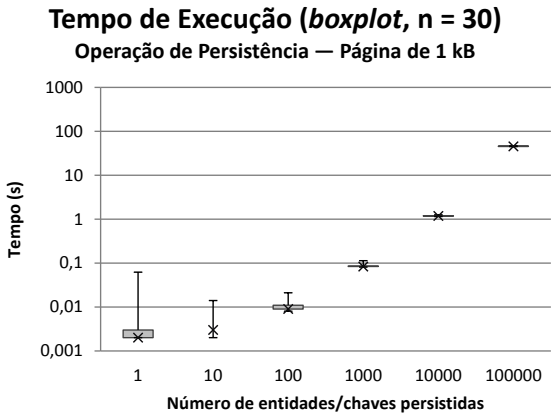


(e) Persistência de Chaves — Página de 4kB

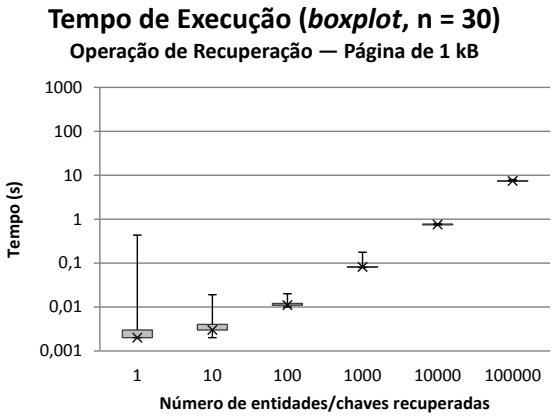


(f) Recuperação de Chaves — Página de 4kB

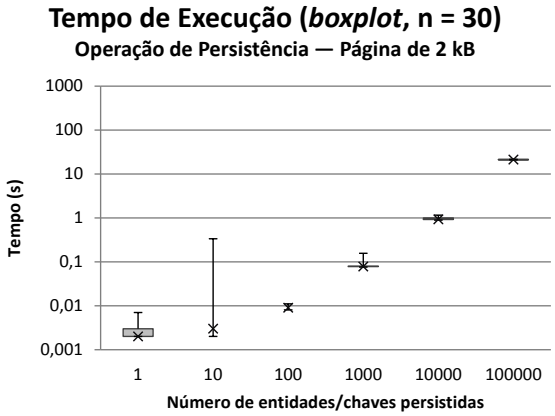
Figura 50: Condicionamento da Atualização em Disco: Número de operações de escrita em disco para persistência e recuperação de chaves.



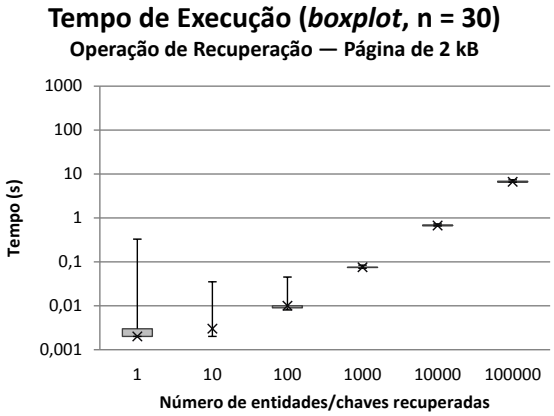
(a) Operação de Persistência — Página de 1kB



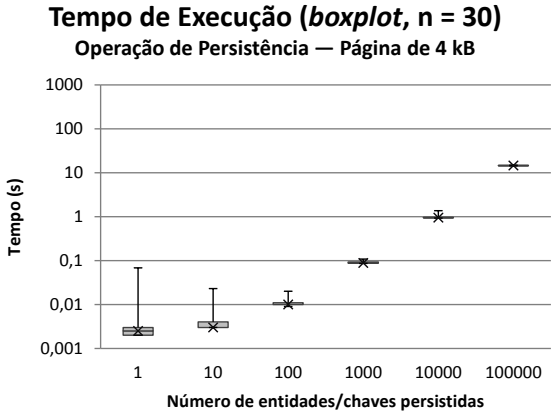
(b) Operação de Recuperação — Página de 1kB



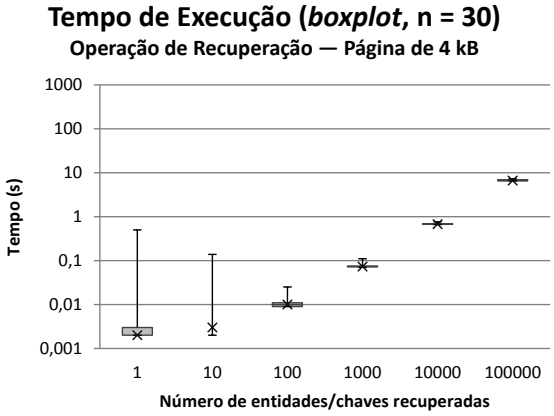
(c) Operação de Persistência — Página de 2kB



(d) Operação de Recuperação — Página de 2kB

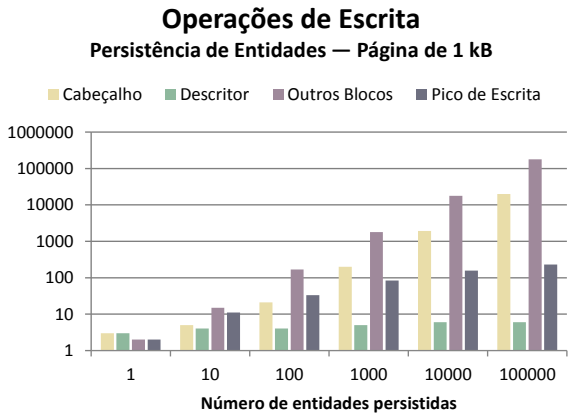


(e) Operação de Persistência — Página de 4kB

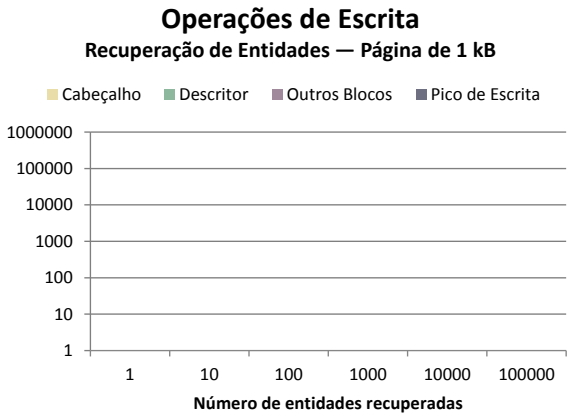


(f) Operação de Recuperação — Página de 4kB

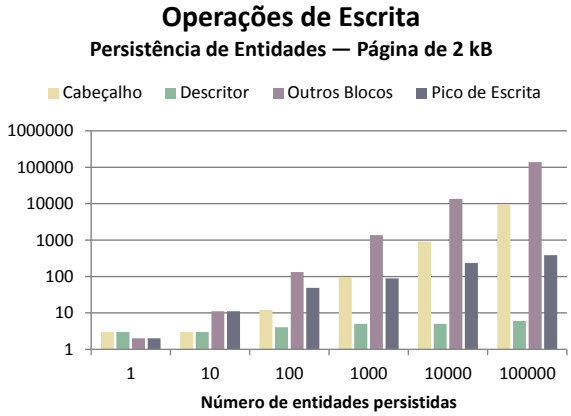
Figura 51: Condicionamento da Atualização do Registro *Last Session ID*: Tempo de execução.



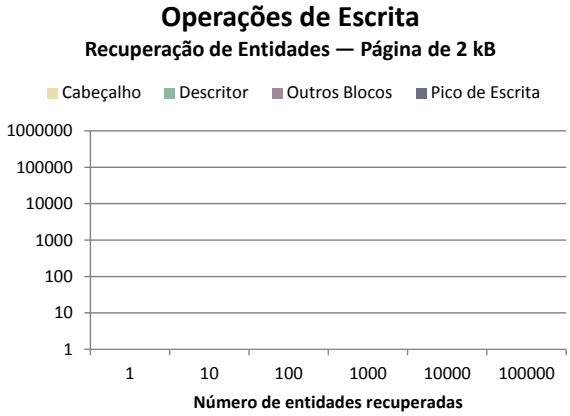
(a) Persistência de Entidades — Página de 1kB



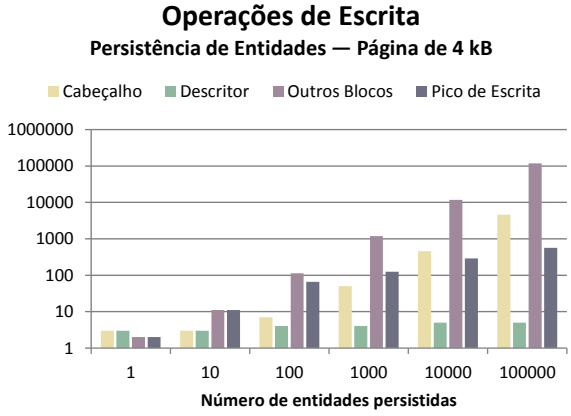
(b) Recuperação de Entidades — Página de 1kB



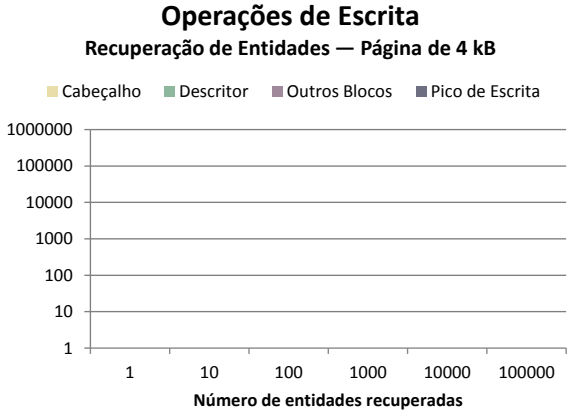
(c) Persistência de Entidades — Página de 2kB



(d) Recuperação de Entidades — Página de 2kB

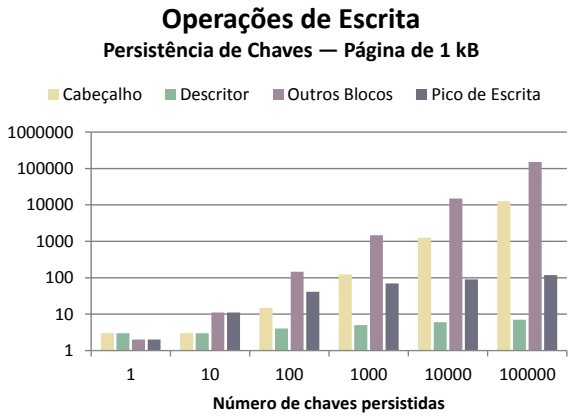


(e) Persistência de Entidades — Página de 4kB

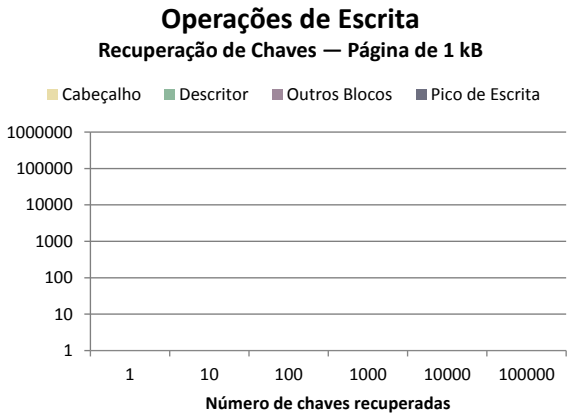


(f) Recuperação de Entidades — Página de 4kB

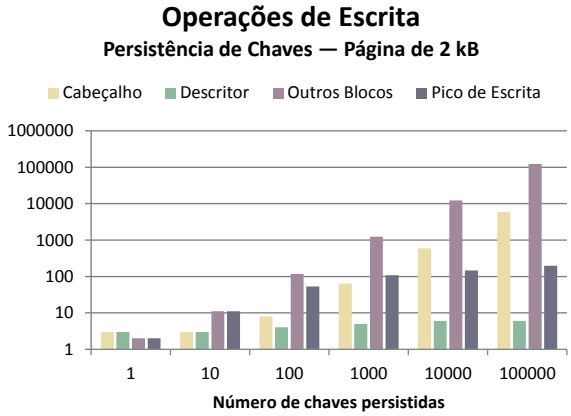
Figura 52: Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco para persistência e recuperação de entidades.



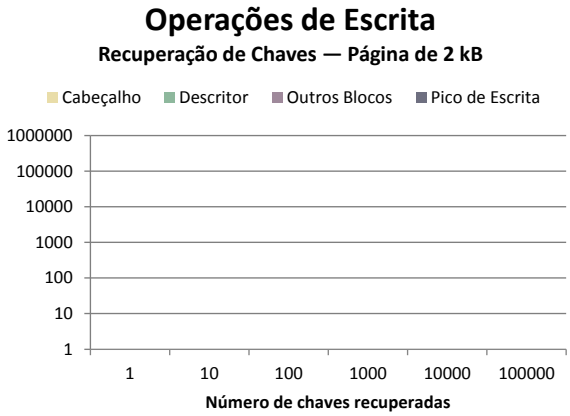
(a) Persistência de Chaves — Página de 1kB



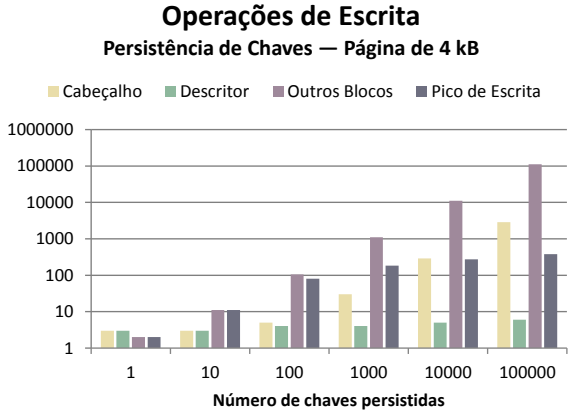
(b) Recuperação de Chaves — Página de 1kB



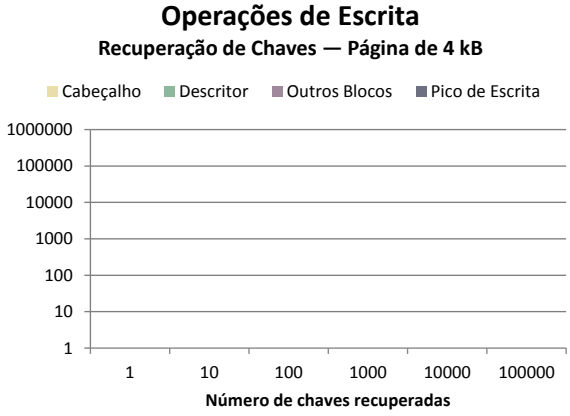
(c) Persistência de Chaves — Página de 2kB



(d) Recuperação de Chaves — Página de 2kB



(e) Persistência de Chaves — Página de 4kB



(f) Recuperação de Chaves — Página de 4kB

Figura 53: Condicionamento da Atualização do Registro *Last Session ID*: Número de operações de escrita em disco para persistência e recuperação de chaves.

Referências

ABADI, D. et al. The beckman report on database research. *ACM SIGMOD Record*, ACM, v. 43, n. 3, p. 61–70, 2014.

BALL, S. *Embedded Microprocessor Systems: Real World Design*. 3. ed. USA: Newnes, 2002. 432 p. ISBN 9780080477572.

BARR, M. *Programming embedded systems in C and C++*. 1. ed. Sebastopol, CA, USA: O'Reilly Media, 1999. 200 p. ISBN 9781565923546.

BOUKHOBZA, J. et al. Embedded databases on flash memories: Performance and lifetime issues, the case of SQLite. In: *Embedded Real-time Software and Systems ERTS*. [S.l.: s.n.], 2014. p. 140.

CARVALHO, L. O. *Object-Injection*: um framework de indexação e persistência. 88 p. Dissertação (Mestrado em Ciência e Tecnologia da Computação) — Universidade Federal de Itajubá, Itajubá, 2013.

Chicago Police Department — Crime Reports — 2001 to present. 2015. Disponível em: <`https://data.cityofchicago.org/Public-Safety/Crimes-2001-to-present/ijzp-q8t2`>. Acesso em: 02 nov 2015.

Chicago Police Department — Illinois Uniform Crime Reporting (IUCR) Codes. 2015. Disponível em: <`https://data.cityofchicago.org/Public-Safety/Chicago-Police-Department-Illinois-Uniform-Crime-R/c7ck-438e`>. Acesso em: 02 nov 2015.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. (VLDB '97), p. 426–435. ISBN 1-55860-470-7. Disponível em: <http://dl.acm.org/citation.cfm?id=645923.671005>.

City of Chicago — Food Inspections. 2015. Disponível em: <`https://data.cityofchicago.org/Health-Human-Services/Food-Inspections/4ijn-s7e5`>. Acesso em: 02 nov 2015.

COOPER, J.; JAMES, A. Challenges for database management in the internet of things. *IETE Technical Review*, Taylor & Francis, v. 26, n. 5, p. 320–329, 2009.

Corel Image Features Data Set. 2015. Disponível em: <`http://archive.ics.uci.edu/ml/datasets/Corel+Image+Features`>. Acesso em: 10 out 2015.

DEITEL, P.; DEITEL, H. *C++: How to Program*. 8. ed. Boston, MA, USA: Prentice Hall, 2011. 1104 p. ISBN 9780132662369.

DOUGLAS, G.; LAWRENCE, R. LittleD: a SQL database for sensor nodes and embedded applications. In: ACM. *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. [S.l.], 2014. p. 827–832.

ELMASRI, R.; NAVATHE, S. *Sistemas de Banco de Dados*. 6. ed. São Paulo: Pearson Addison Wesley, 2011. 724 p. ISBN 9788579360855.

Firebird. 2014. Disponível em: <`http://www.firebirdsql.org/`>. Acesso em: 15 mar 2014.

FreeRTOS™ — Memory Management. 2015. Disponível em: <`http://www.freertos.org/a00111.html`>. Acesso em: 01 set 2015.

FreeRTOS™ Features. 2015. Disponível em: < `http://www.freertos.org/FreeRTOS_Features.html`>. Acesso em: 01 set 2015.

FreeRTOS™ Windows Port. 2015. Disponível em: <`http://www.freertos.org/FreeRTOS-Windows-Simulator-Emulator-for-Visual-Studio-and-Eclipse-MingW.html`>. Acesso em: 01 set 2015.

FreeRTOS™+CLI. 2015. Disponível em: < `http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_CLI/FreeRTOS_Plus_Command_Line_Interface.shtml`>. Acesso em: 03 set 2015.

GAL, E.; TOLEDO, S. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, ACM, v. 37, n. 2, p. 138–163, 2005.

GANSSLE, J. G. *The Art of Designing Embedded Systems*. 1. ed. Newton, MA, USA: Butterworth-Heinemann, 1999. 256 p. ISBN 9780750698696.

GHILARDELLI, A.; CORNO, S. Flash cards. In: *Inside NAND Flash Memories*. Springer Netherlands, 2010. p. 483–513. ISBN 9789048194308. Disponível em: <http://dx.doi.org/10.1007/978-90-481-9431-5_17>.

GOSLING, J. et al. *The Java® Language Specification — Java SE 8 Edition*. Mar 2014. 780 p. Disponível em: <http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. In: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 1984. (SIGMOD '84), p. 47–57. ISBN 0-89791-128-8. Disponível em: <http://doi.acm.org/10.1145/602259.602266>.

HOOK, B. *Write portable code: an introduction to developing software for multiple platforms*. 1. ed. San Francisco, CA, USA: No Starch Press, 2005. 272 p. ISBN 9781593270568.

HYDE, R. *Write Great Code, Vol. 2: Thinking Low-Level, Writing High-Level*. 1. ed. San Francisco, CA, USA: No Starch Press, 2006. 640 p. ISBN 9781593270650.

IACULO, M. et al. Memory cards. In: MICHELONI, R.; CAMPARDO, G.; OLIVO, P. (Ed.). *Memories in Wireless Systems*. Springer Berlin Heidelberg, 2008, (Signals and Communication Technology). p. 67–93. ISBN 9783540790778. Disponível em: <http://dx.doi.org/10.1007/978-3-540-79078-5_4>.

IEEE Std 610. IEEE standard computer dictionary: A compilation of IEEE standard computer glossaries. p. 1–217, Jan 1991.

IEEE Std 754. IEEE standard for floating-point arithmetic. p. 1–70, Aug 2008. Disponível em: <http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=4610935>.

ISO/IEC. *ISO/IEC 9899:201x Programming languages — C (Committee Draft)*. Geneva, Switzerland: International Organization for Standardization, 2011. 701 p. Disponível em: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>.

ISO/IEC. *Working Draft, Standard for Programming Language C++*. Geneva, Switzerland: International Organization for Standardization, 2014. 1354 p. Disponível em: <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4296.pdf>.

JIMNEZ, M.; PALOMERA, R.; COUVERTIER, I. *Introduction to Embedded Systems: Using Microcontrollers and the MSP430*. [S.l.]: Springer Publishing Company, Incorporated, 2013. ISBN 1461431425, 9781461431428.

KANG, W. et al. QeDB: A quality-aware embedded real-time database. In: IEEE. *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*. [S.l.], 2009. p. 108–117.

KANG, W. et al. Design, implementation, and evaluation of a qos-aware real-time embedded database. *Computers, IEEE Transactions on*, IEEE, v. 61, n. 1, p. 45–59, 2012.

KARLSSON, J. S. et al. IBM DB2 everyplace: A small footprint relational database system. In: IEEE. *ICCCN*. [S.l.], 2001. p. 0230.

KIM, G.-J. et al. LGeDBMS: a small DBMS for embedded system with flash memory. In: VLDB ENDOWMENT. *Proceedings of the 32nd international conference on Very large data bases*. [S.l.], 2006. p. 1255–1258.

KOLTSIDAS, I.; VIGLAS, S. D. Data management over flash memory. In: ACM. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. [S.l.], 2011. p. 1209–1212.

LINDHOLM, T. et al. *The Java® Virtual Machine Specification — Java SE 8 Edition*. Feb 2015. 604 p. Disponível em: <http://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>.

LOGAN, S. *Cross-platform Development in C++: Building Mac OS X, Linux, and Windows Applications*. 1. ed. Boston, MA, USA: Addison-Wesley Professional, 2007. 576 p. ISBN 9780321246424.

LPCXPRESSO Application Flash / RAM size. 2015. Disponível em: <`https://www.lpcware.com/content/faq/lpcxpresso/application-flash-ram-size`>. Acesso em: 10 dez 2015.

MADDEN, S. R. et al. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on database systems (TODS)*, ACM, v. 30, n. 1, p. 122–173, 2005.

MERRITT, R. *Slideshow: 10 Embedded Design Trends*. 2014. Disponível em: <http://www.eetimes.com/document.asp?doc_id=1322014>. Acesso em: 10 set 2015.

MICHELONI, R.; MARELLI, A.; COMMODARO, S. NAND overview: from memory to systems. In: *Inside NAND Flash Memories*. Springer Netherlands, 2010. p. 19–53. ISBN 9789048194308. Disponível em: <http://dx.doi.org/10.1007/978-90-481-9431-5_2>.

Microsoft SQL Server Compact. 2015. Disponível em: < https://www.microsoft.com/en-us/download/details.aspx?id=17876>. Acesso em: 19 ago 2014.

NATH, S.; KANSAL, A. FlashDB: dynamic self-tuning database for NAND flash. In: ACM. *Proceedings of the 6th international conference on Information processing in sensor networks*. [S.l.], 2007. p. 410–419.

NORI, A. Mobile and embedded databases. *IEEE Data Engineering Bulletin Issues*, v. 30, n. 3, p. 3–12, Setembro 2007. Disponível em: <http://sites.computer.org/debull/A07sept/nori.pdf>. Acesso em: 08 out 2014.

NULL, L.; LOBUR, J. *Essentials of Computer Organization and Architecture*. 3. ed. USA: Jones and Bartlett Publishers, Inc., 2003. 673 p. ISBN 978-0763704445.

Oracle Berkeley DB. 2014. Disponível em: <https://oss.oracle.com/berkeley-db.html>. Acesso em: 20 ago 2014.

Oracle Database Lite Client 10g. 2014. Disponível em: <http://www.oracle.com/technetwork/database/database-lite/lite-client-090611.html>. Acesso em: 20 ago 2014.

ORTIZ, S. Embedded databases come out of hiding. *Computer*, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 33, n. 3, p. 16–19, mar. 2000. ISSN 0018-9162. Disponível em: <http://dx.doi.org/10.1109/MC.2000.825689>.

PRATA, S. *C Primer Plus*. 6. ed. [S.l.]: Pearson Education, 2013. 1080 p. ISBN 9780133432381.

PUCHERAL, P. et al. PicoDBMS: Scaling down database techniques for the smartcard. *The VLDB Journal*, Springer, v. 10, n. 2-3, p. 120–132, 2001.

QIAN, K.; HARING, D. D.; CAO, L. *Embedded Software Development with C*. 1. ed. New York, NY, USA: Springer US, 2009. 390 p. ISBN 9781441906069.

Radiant Insights. *Global embedded system market size worth* $214.39 *billion by 2020*. 2015. Disponível em: <http://www.radiantinsights.com/press-release/global-embedded-system-market>. Acesso em: 29 fev 2016.

ROSENMÜLLER, M. et al. Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data & Knowledge Engineering*, Elsevier, v. 68, n. 12, p. 1493–1512, 2009.

ROSENMÜLLER, M. et al. Specialized embedded DBMS: Cell based approach. In: IEEE. *Database and Expert Systems Application, 2009. DEXA'09. 20th International Workshop on*. [S.l.], 2009. p. 9–13.

SAAKE, G. et al. Downsizing data management for embedded systems. *Egyptian Computer Science Journal*, v. 31, n. 1, p. 1–13, 2009.

SIEGMUND, N. et al. Towards robust data storage in wireless sensor networks. *IETE Technical Review*, Taylor & Francis, v. 26, n. 5, p. 335–340, 2009.

SILBERSCHATZ, A.; KORTH, H.; SUDARSHAN, S. *Database System Concepts*. 6. ed. New York, NY, USA: McGraw-Hill Education, 2011. 1376 p. ISBN 9780073523323.

SQLite. 2014. Disponível em: <`http://www.sqlite.org`>. Acesso em: 15 mar 2014.

Super Lean FAT File System. 2015. Disponível em: < `http://www.freertos.org/FreeRTOS-Plus/FreeRTOS_Plus_FAT_SL/FreeRTOS_Plus_FAT_SL.shtml`>. Acesso em: 03 set 2015.

THE UNICODE CONSORTIUM. *The Unicode Standard, Version 7.0.0*. Mountain View, CA: The Unicode Consortium, 2014. ISBN 978-1-936213-09-2. Disponível em: <http://www.unicode.org/versions/Unicode7.0.0/>.

TSIFTES, N.; DUNKELS, A. A database in every sensor. In: ACM. *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*. [S.l.], 2011. p. 316–332.

VALGRIND. 2014. Disponível em: <`http://www.valgrind.org`>. Acesso em: 04 ago 2014.

WANG, J. et al. Block-based multi-version B-Tree for flash-based embedded database systems. *Computers, IEEE Transactions on*, IEEE, v. 64, n. 4, p. 925–940, 2015.

WHANG, K.-Y. et al. The ubiquitous DBMS. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 38, n. 4, p. 14–22, jun. 2010. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/1815948.1815952>.

YADAVA, H. *The Berkeley DB Book*. Berkely, CA, USA: Apress, 2007. 442 p. ISBN 9781590596722.