JDIET
FOOTPRINT REDUCTION FOR MEMORY-CONSTRAINED SYSTEMS

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Michael J. Huffman

June 2009

©2009

Michael John Huffman

ALL RIGHTS RESERVED

APPROVAL PAGE

TITLE: JDiet: Footprint Reduction for Memory-constrained Systems

AUTHOR: Michael J. Huffman

DATE SUBMITTED: June 2009

Dr. Alexander Dekhtyar
Advisor or Committee Chair

Dr. John Clements
Committee Member

Dr. David Janzen
Committee Member

Abstract

JDiet: Footprint Reduction for Memory-constrained Systems

by

Michael J. Huffman

Main memory remains a scarce computing resource. Even though main memory is becoming more abundant, software applications are inexorably engineered to consume as much memory as is available. For example, expert systems, scientific computing, data mining, and embedded systems commonly suffer from the lack of main memory availability.

This thesis introduces JDiet, an innovative memory management system for Java applications. The goal of JDiet is to provide the developer with a highly configurable framework to reduce the memory footprint of a memory-constrained system, enabling it to operate on much larger working sets. Inspired by buffer management techniques common in modern database management systems, JDiet frees main memory by evicting non-essential data to a disk-based store. A buffer retains a fixed amount of managed objects in main memory. As non-resident objects are accessed, they are swapped from the store to the buffer using an extensible replacement policy.

While the Java virtual machine naïvely delegates virtual memory management to the operating system, JDiet empowers the system designer to select both the managed data and replacement policy. Guided by compile-time configuration, JDiet performs aspect-oriented bytecode engineering, requiring no explicit cou-

pling to the source or compiled code.

The results of an experimental evaluation of the effectiveness of JDiet are reported. A JDiet-enabled XML DOM parser is capable of parsing and processing over 200% larger input documents by sacrificing less than an order of magnitude in performance.

Keywords

Buffer management, virtual memory, memory management, object loitering, orthogonal persistence, bytecode, database

Categories

D.1 [**Software**]: Programming Techniques – Aspect-Oriented Programming

D.3.2 [**Programming Languages**]: Language Classifications – Java

D.4.2 [**Operating Systems**]: Storage Management – Virtual Memory

H.2.4 [**Database Management**]: Systems – Buffer Management

H.2.3 [**Database Management**]: Languages – Database (Persistent) Programming Languages

For Krystle

Table of Contents

List of Tables	xi
List of Figures	xii
List of Algorithms	xiii
1 Introduction	1
1.1 Overview	5
1.2 Notation & Modeling	5
2 Background	6
2.1 Memory Management	6
2.1.1 Hierarchical Memory	6
2.1.2 Virtual Memory	7
2.1.3 Explicit & Implicit Management	10
2.2 Buffer Management	13
2.2.1 Reference Behavior	15
2.2.2 Page Replacement Policies	16
2.3 Java	21
2.3.1 Object Model	21
2.3.2 Reference Types	22
2.3.3 Runtime Data Areas	23
2.3.4 Java Object Serialization	25
2.3.5 Bytecode	26
2.4 Aspect-oriented Programming	28
2.4.1 Weaving	31
2.4.2 Discussion	31
3 Problem	33
4 Solution	36
4.1 Design Principles	38
4.2 Approaches	39
4.2.1 Virtual machine alteration	40
4.2.2 Reflection	40

4.2.3	Bytecode engineering	41
4.2.4	Source alteration	42
4.2.5	Selected Approach	42
4.3	Architecture	43
4.3.1	Compile-time Weaver	43
4.3.2	Runtime Manager	46
4.4	Implementation	53
4.4.1	Logging	54
4.4.2	Runtime Manager	54
4.4.3	Compile-team Weaver	73
5	Evaluation	86
5.1	Benchmark	87
5.2	Experimental Design	88
5.2.1	Independent Variables	89
5.2.2	Dependent Variables	90
5.2.3	Hardware Configuration	91
5.3	Results	91
5.3.1	32 MB Maximum Heap Size	91
5.3.2	256 MB Maximum Heap Size	94
5.3.3	1024 MB Maximum Heap Size	98
5.3.4	Analysis	102
5.3.5	Discussion	104
6	Related Work	106
6.1	Memory Leak Tolerance	106
6.1.1	Melt	108
6.1.2	LeakSurvivor	109
6.1.3	PANACEA	110
6.1.4	Discussion	111
6.2	Databases	113
6.3	Orthogonal Persistence	115
6.3.1	Persistent Store	117

6.3.2	Implementation Approaches	117
6.3.3	Discussion	125
6.4	External Memory Algorithms	128
6.5	Footprint Reduction, Compression & Optimization	128
7	Conclusion	130
7.1	Future Work	132
7.1.1	Collections	132
7.1.2	Variability & Precision	133
7.1.3	Object-model Test Suite	133
7.1.4	Eviction Policies	134
8	Appendices	135
A	XMark Benchmark File Characteristics	135
B	Configuration XML Schema	136
C	Bytecode Example	137
D	Java Virtual Machine Instructions	138
E	UML Diagrams	141
F	XMark Benchmark File Characteristics	143
F.1	32 MB Heap Size	143
F.2	256 MB Heap Size	143
F.3	1024 MB Heap Size	145
	Glossary	151
	Index	154
	References	160

List of Tables

1	Descriptor types	26
2	Heuristic Data Type Weights	80
3	XMark benchmark file characteristics	135
4	Parse Memory Usage with 32 MB Max. Heap Size	144
5	Parse Total Time with 32 MB Max. Heap Size	144
6	Depth-first iteration Total Time with 32 MB Max. Heap Size . .	145
7	Parse Memory Usage with 256 MB Max. Heap Size	146
8	Parse Total Time with 256 MB Max. Heap Size	146
9	Depth-first Iteration Total Time with 256 MB Max. Heap Size . .	147
10	Parse Memory Usage with 1024 MB Max. Heap Size	148
11	Parse Total Time with 1024 MB Max. Heap Size	149
12	Depth-first Iteration Total Time with 1024 MB Max. Heap Size .	150

List of Figures

1	Tiers in the memory hierarchy	7
2	JDiet subsystems	44
3	Compile-time Weaver component architecture	45
4	Runtime Manager interaction with thin and fat classes	47
5	Runtime Manager component architecture	48
6	Compile-time Weaver workflow	77
7	Parse Memory Usage with 32 MB Max. Heap Size	92
8	Parse Total Time with 32 MB Max. Heap Size	93
9	Iteration Total Time with 32 MB Max. Heap Size	94
10	Parse Memory Usage with 256 MB Max. Heap Size	96
11	Parse Total Time with 256 MB Max. Heap Size	97
12	Iteration Total Time with 256 MB Max. Heap Size	98
13	Parse Memory Usage with 1024 MB Max. Heap Size	99
14	Parse Total Time with 1024 MB Max. Heap Size	100
15	Iteration Total Time with 1024 MB Max. Heap Size	101
16	Compile-time Weaver class design	141
17	Runtime Manager class design	142

List of Algorithms

1	CTW fat extraction	46
2	Buffer Manager: <code>manage</code>	49
3	Buffer Manager: <code>load</code>	49
4	Buffer: <code>insert</code>	51
5	Buffer: <code>find</code>	52

1 Introduction

Main memory is a scarce system resource, despite advances in chip design and lowering costs. There is inevitably a strong correlation between software and hardware development; as hardware increases its computational and storage capacity, software is inexorably engineered to exploit these emergent capabilities. Even though more main memory is made available to systems, software constructed to run on these systems demand more memory. Two communities exist which encounter two distinct main memory obstacles. In the first community, cost dictates memory availability. Contrary to popular belief, the cost of main memory still dominates total system cost [Ekman and Stenstrom, 2004; Hennessy and Patterson, 1996; Mahapatra and Venkatrao, 1999]. Rather than cost, the second community is often limited due to energy consumption [Azimi et al., 2007; Benini et al., 2003; Chen et al., 2003; Graefe, 2007; Shim et al., 2003]. Ekman and Stenstrom [2004] forecast that costs, physical space, and energy consumption of main memory will be a limiting factor in the near future. Multitasking systems exacerbate this problem [Choi and Han, 2008]. In multitasking computing systems a single process is unlikely to be granted all available memory, as it must be shared by multiple concurrently executing software processes.

A diverse set of software systems suffer from limited main memory [Becerra et al., 2003; Bhadra et al., 2009]. Examples of memory-constrained systems include expert systems [Marney and Ibrahim, 1995], scientific computing [Zhang et al., 2005], data mining [Kantardzic, 2002; Liu, 2008], business applications, and sensor networks [Liu et al., 2005]. Because of its safety and portability, Java is commonly utilized to implement software on resource-constrained devices [Bhadra et al., 2009; Zhang and Krintz, 2005]. For example, the K virtual machine

(KVM) [Sun Microsystem, Inc., 2000] is engineered for running Java applications in embedded systems with less than 128 KB of main memory. Software written for deployment in such resource-constrained environments would benefit from a generic solution addressing the limited availability of main memory. In particular, existing memory management techniques and tools do not fully address the needs of object-oriented memory-constrained systems.

Several techniques can be used to allow memory-constrained systems to operate on data sets larger than the size of main memory, but from the author's perspective these techniques suffer from lack of reusability, poor modularity, cross-cutting concerns, limited configurability, and costly maintenance. While virtual memory can provide the illusion of infinite memory [Denning, 1970], relying purely on virtual memory is suboptimal, as the virtual memory manager (typically an operating system) is unable to make intelligent swapping decisions [Azimi et al., 2007]. Memory leak tools [Bond and McKinley, 2006; Breitgand et al., 2007; Goldstein et al., 2007; Hauswirth and Chilimbi, 2004; Jump and McKinley, 2007; Tang et al., 2008] are effective in detecting and tolerating memory leaks, but do not address systems that may have insufficient main memory available in the first place. These tools presume that developer negligence is the crux of main memory exhaustion rather than the physical availability.

This thesis is based on the observation of *temporal cruciality*, that access to some parts of objects can be more time-critical than others. Prime attributes are attributes that are accessed frequently and require the fastest access times. On the other hand, non-prime attributes are infrequently accessed and may be subject to slower access times. Depending on the system and any user-interfaces, access to prime attributes is performed much more frequently than non-prime attributes. This disparity is readily apparent in information-rich systems, such

as those employing an ontology [Devedzić, 2002]. *Temporal cruciality* is strongly correlated with *intra-object locality*, an extension of the well-known principle of *locality*. *Intra-object locality* refers to the natural tendency of a subset of an object’s attributes to be used at the same time. *Intra-object locality* is commensurate with *temporal cruciality*, as the subset of attributes accessed most frequently are intuitively considered most critical.

There exists no means by which system designers can natively encode such considerations into the Java programming language. Like many other object-oriented languages, Java presumes that all data should be accessible at any time in main memory. In other words, all attributes are equally critical. Consequently, persistence technologies such as Java Data Objects [Sun Microsystems, Inc., 2009b], Enterprise JavaBeans [Sun Microsystems, Inc., 2009a], and Hibernate [Red Hat Middleware, 2009], have been developed to facilitate the configurable persistence of Java objects to auxiliary memory. However, such tools require considerable changes to the application code and these changes introduce a cross-cutting concern that permeates the system. Development using these tools requires additional cognitive burden on the system designers, as eviction decisions must be made in each cross-cutting location. This can lead to situations in which the decisions made in each location are inconsistent. Persistent programming languages, such as PJava [Atkinson, 2001], address this cross-cutting concern by providing orthogonal persistence. However, the principle of persistence independence [Atkinson and Morrison, 1995] prohibits the developer from identifying the persistent entities explicitly, nor tuning the persistence mechanism to meet the needs of the system designer.

This thesis introduces JDiet, an innovative semi-automated memory management system for memory footprint reduction. The goal of JDiet is to provide

the developer with a highly configurable framework to reduce the memory requirements of memory-constrained systems. JDiet delegates the responsibility of identifying what data is least time-critical to the system designer. The attributes (fields) of classes which are deemed less critical are referred to as *fat attributes*. The choice of *fat attributes* requires a thorough understanding of the system JDiet is managing, as *fat attributes* under the purview of JDiet are subject to eviction from main memory. In such circumstances, access times for evicted *fat attributes* increase, as disk I/O is required to retrieve the data. JDiet therefore relies on the system designer to perform a memory-performance software engineering tradeoff analysis when selecting *fat attributes*.

Written for Java, JDiet is composed of two sub-systems. The JDiet Compile-time Weaver (CTW) performs byte-code weaving and construction during a post-compile process to *thin* a class into two classes, a *thin class* and *fat class*, which are semantically equivalent to the original class. *Fat objects*, which contain all *fat attributes* from the thinned class, are subsequently managed at runtime by the JDiet Runtime Manager (RM). The RM reacts to main memory pressure by transparently evicting *fat objects* to a persistent storage system. The RM facilitates access from *thin objects* to their respective *fat object*. Unlike similar systems, JDiet runs natively within the Java HotSpot Virtual Machine. Employing aspect-oriented principles, JDiet transparently manages the *fat objects*, with no explicit changes to the source or compiled code.

JDiet is constructed with extensible sub-components, allowing customization for the needs of virtually any memory-constrained system. The system designer may choose to leverage the built-in default implementations or implement customized versions meeting the specific requirements of the system. The JDiet prototype implementation supports three memory replacement policies: (1)

least-recently-used (LRU) (2) first-in first-out (FIFO) and (3) largest-object-first (LOF). While LRU and FIFO are common eviction techniques in database buffer management [Effelsberg and Haerder, 1984], LOF is a novel policy that evicts fat objects based on a heuristic size.

1.1 Overview

The remainder of this thesis is structured as follows. §2 reviews concepts, terminology, and technical references for those lacking familiarity in this area of research. §3 explores memory-constrained systems and their unique problems. §4 outlines the novel solution offered to address these limitations and illuminates the technical approach of JDiet. An experimental design and evaluation of JDiet is presented in §5. §6 highlights related work and existing approaches to solve these problems, as well as their limitations. Lastly, §7 summarizes the contributions of this thesis and draws conclusions about the validity of the proposed solution based on the evaluation.

1.2 Notation & Modeling

Key terms, concepts, and definitions are highlighted with *italicized* text. Classes, types and variables are written in a **true-type** font when presented in text. Textual values of precise content are specified using the start quote (“) and end quote (”) symbols. Code listings are provided for code snippets. In code listings with templated arguments, the left square bracket ([) and right square bracket (]) symbols are used to denote variable input. The Unified Modeling Language (UML) [Object Management Group, 2009] is used to model the conceptual architecture and physical implementation.

2 Background

2.1 Memory Management

An understanding of common memory management techniques is paramount to addressing the issues encountered by memory-constrained systems. §2.1.1 discusses how memory is traditionally organized hierarchically. §2.1.2 describes virtual memory as a mechanism to allow application developers to not concern themselves with hierarchical memory and work with all tiers in the memory hierarchy as if they were a single unit. §2.1.3 contrasts implicit and explicit memory management in order to highlight the benefits and costs of automated memory management. Managed languages use garbage collection, summarized in §2.1.3.1, as a means to automatically reclaim unused memory.

2.1.1 Hierarchical Memory

It is widely accepted that general-purpose computing systems are composed of several levels of memory, ranging from volatile high-speed microprocessor (CPU) registers to slower non-volatile magnetic disks [Brodal and Fagerberg, 2003; Denning, 1970; Vitter, 2001], as illustrated in Figure 1. This tiered design is necessitated by economic factors [Vitter, 2001]. Main memory is limited but fast while auxiliary memory is large but slow¹. However, all data that is accessed by a microprocessor must be placed in its internal registers. Therefore, any data in primary storage or below must be moved to the microprocessor tier, which can be a costly process. Ekman and Stenstrom [2004] present a case for breaking

¹While distribution techniques such as clustering can increase the throughput (bandwidth) of secondary storage [Kunkle and Cooperman, 2008], such a configuration is currently cost-prohibitive for most applications.

memory into even more hierarchical levels.

Figure 1: Tiers in the memory hierarchy

The goal of all memory management systems is to execute an application with as limited a memory footprint as feasible while maintaining program semantics [Yu et al., 2008]. In other words, memory management systems are an effort to optimize the implementation of a storage allocation problem solution.

Definition 1. storage allocation problem: *The problem of determining what data is resident in each part of the memory hierarchy at any time.* [Denning, 1970]

2.1.2 Virtual Memory

Modern software often has demanding memory requirements [Chin et al., 2004]. As main memory is a shared resource, an application may only use a subset of the available physical memory, particularly in multitasking systems. To ameliorate this problem, virtual memory allows an application to be allocated more memory than exists in physical main memory.

Virtual memory has become ubiquitous in modern operating systems. Virtual memory divides memory into a virtual address space and memory space [Denning, 1970]. Doing so provides applications the illusion of a much larger address

space [Appel and Li, 1991]. The virtual address space composes those addresses (or identifiers) that an application uses to reference memory. The memory space composes the locations in which the elements in memory actually reside. The translation of virtual address to memory address is accomplished by the translation lookaside buffer (TLB) [Hirzel, 2007]. Virtual memory allows decoupling of memory access from machine dependent memory configuration and location.

Definition 2. virtual memory: *Façade that provides an illusion of contiguous main memory where in fact such main memory may be fragmented or offloaded to auxiliary memory.*

If a software system request more memory than is physically available, most modern operating systems allocate into a swap space on disk. The swap space resides in auxiliary memory, but using virtual memory, the system is unaware of the true location of the data. The main memory allocated then acts as a fixed size cache for all data required by an application. When data accessed by an application does not reside in main memory, the operating system automatically swaps auxiliary memory into main memory and fulfills the request. The part of the operating system used to facilitate this memory movement is known as the Memory Management Unit (MMU) [Appel and Li, 1991].

Virtual memory, however, incurs a significant performance penalty if the virtual memory manager chooses to allocate a swap space on disk, as doing so imposes significant degradations to overall system performance. Consequently, the replacement policy used by the operating system to determine which data elements to swap, and when, is vital. However, general operating systems must make assumptions about reference behavior that apply to all applications. Common wisdom holds that generic solutions, such as the choice of operating system

page replacement policy, often lead to much poorer performance than solutions engineering with deeper understanding of reference behavior.

When facing memory pressure, applications can exhibit thrashing—the tendency of an application to cyclically substitute data from auxiliary memory to main memory. Applications that exhibit virtual memory thrashing are performance bound to auxiliary memory speeds, which are several orders of magnitude slower than main memory². When coupled with most buffer managers in modern databases, this can also lead to the double paging problem [Brice and Sherman, 1977; Chew and Silberschatz, 1992; Fernandez et al., 1978; Goldberg and Haslinger, 1974; Lang et al., 1977; Sherman and Brice, 1976a,b; Tuel, 1976].

To improve efficiency of virtual memory, adjacent blocks of memory are grouped into pages. Pages optimize the transfer of data from auxiliary memory to main memory, as most auxiliary memory devices are engineered to read blocks of data in a single read operation. Pages are the fundamental unit of data swapped between auxiliary and main memory. Paging is commonly understood to be far superior to non-paging, even though it leads to fragmentation [Denning, 1970]. Fragmentation occurs when an application is allotted more memory than requested because the request size is rounded up to the nearest page size. However, intelligent selection of the page size can combat fragmentation. The size of a page is determined by the operating system based on anticipated fragmentation and the efficiency of auxiliary to main memory transport.

Definition 3. page hit: *The situation in which a memory page being accessed is resident in main memory.*

Definition 4. page miss: *The situation in which a memory page being accessed is not resident in main memory and must therefore be swapped from auxiliary*

²Approximately six orders [Hertz et al., 2005].

memory to main memory.

2.1.3 Explicit & Implicit Management

Memory management can be handled either explicitly or implicitly by the runtime environment. Languages supporting these techniques are referred to as unmanaged and managed languages, respectively. In an unmanaged language, it is the responsibility of the programmer to allocate memory prior to its use and deallocate once finished. Common unmanaged languages include Pascal, C, and C++. On the other hand, managed languages provide a memory management system that automatically allocates memory and frees unused memory whenever possible, through techniques such as garbage collection. Both Java and C# are managed languages with the Java virtual machine and Common Language Runtime providing the automated memory management respectively. Other managed languages include Smalltalk, ML, Self, Modula-3, and Eiffel.

Debate remains as to the software engineering benefits of one approach over the other. However, common wisdom dictates that automated memory management can reduce memory leaks, cognitive burden on developers, and improve safety. [Phipps \[1999\]](#) conducted a rudimentary study of the bug rates in managed vs unmanaged code. Although the study demonstrated that observed bug rates in C++ were nearly 3 times as high as Java, the statistical basis for the study merits caution in applying these results to a greater population. The study included a single program ported from Java to C++ in which the experimenter was the subject and admittedly was more familiar in Java.

The primary benefit of explicit memory management is performance. While [Appel \[1987\]](#) argues that given enough memory, managed memory can be faster

than explicit memory management, [Wilson et al. \[1992\]](#) argue this conclusion is unlikely to hold for modern languages with deep, complex memory structures. Because of the disparity between main memory and auxiliary memory speeds, *locality* effects dominate performance [[Wilson et al., 1992](#); [Yu et al., 2008](#)]. Consequently, the manner in which memory is allocated can significantly affect runtime performance. [Hertz and Berger \[2005\]](#) conclude that automatic memory management (specifically garbage collection) hinders performance by an average of 17% and requires significantly more memory to operate efficiently, confirming similar results of [Steele \[1975\]](#), [Ungar \[1984\]](#), and [Diwan et al. \[1995\]](#).

2.1.3.1 Garbage Collection

Garbage collection has formally been defined as “the automatic reclamation of computer storage” [[Knuth, 1969](#)] and is the foundation for the memory management system of managed languages. The goal of garbage collection is to reduce memory related errors, such as dangling pointers [[Printezis, 1996](#)]. Using reachability analysis, all garbage collectors periodically search through allocated memory to find blocks of memory a program can no longer use. Memory is reachable if it can be accessed directly or indirectly through one or more dereference operations, starting from some discrete set of global or static memory blocks. The data set reachable at any point in time on the heap is referred to as the residency [[Jones and Ryder, 2008](#)]. Once the residency has been identified, the garbage collector frees unused memory without affecting the program’s future execution [[Shaham et al., 2001](#)].

Definition 5. garbage collection: *The automatic reclamation of computer storage. [[Knuth, 1969](#)]*

Morrisett et al. [1995] present λ_{gc} as a formalization of the semantics of garbage collection, based on λ -calculus. λ_{gc} formalizes the key operations and semantics common to all garbage collectors. Each garbage collection system is focused on (1) the method of memory allocation (2) the triggers that cause the collector to execute and (3) determining when, and if, the heap should be expanded to accommodate new memory [Brecht et al., 2006]. The choice of heap size is particularly critical to most garbage collectors, as by their nature they are sensitive to the size of the heap. Garbage collection performance can improve up to ten-fold with increases in the heap size [Yang et al., 2006]. The frequency of collection is inversely related to the size of the heap [Appel, 1987].

Most garbage collection strategies are reactive, rather than preemptive [Yang et al., 2006]. Instead of collecting early and often, they typically wait until the heap is full and then decide whether to perform a collection. At some point in time when the allocated memory fills the current heap, the garbage collector has one of two strategies to employ: (1) collect immediately to avoid allocating into virtual memory and costly paging or (2) continue allocating through virtual memory. Taking the former approach limits the use of virtual memory and restricts the heap to sizes that will fit in the available main memory. The latter approach does not impose such a restriction, but may lead to unacceptable performance costs depending on the reference behavior of the application. Allocating within virtual memory can lead to damaging swapping behavior and thrashing [Hertz et al., 2005]. With the increasing popularity of managed languages, a vast quantity of garbage collection techniques have been proposed. A programmer must understand the tradeoffs of each algorithm to optimize performance for a particular application [Hirzel et al., 2001]. Wilson [1992] provides a comprehensive, survey of uniprocessor garbage collection techniques.

The primary drawback of garbage collection is performance. Advances in Just-in-Time (JIT) compiler techniques are offsetting traditional performance slowdowns in most garbage collected languages [Rayside and Mendel, 2007]. Managed languages are becoming more popular because they can reduce the amount of memory related bugs [Huang et al., 2004]. Unmanaged languages are unable to modify the data layout at runtime³, as it must use a non-moving allocator such as a free list. Consequently, allocation and deallocation in unmanaged languages often leads to fragmentation and poor *locality* [Wilson et al., 1992; Yu et al., 2008]. On the other hand, managed languages may utilize garbage collectors that move the location of objects to increase efficiency, such as copying and/or compacting collectors [Siegwart and Hirzel, 2006].

2.2 Buffer Management

The buffer manager component of a modern Database Management System (DBMS) is “maintained for the purposes of interfacing main memory and disk” [Effelsberg and Haerder, 1984]. The buffer manager is responsible for reading and writing all changes to the data in the database, and uses buffers to cache a subset of all the data (often referred to as a working set). The goal of all buffer managers is to minimize disk access and maximize throughput [Cai et al., 1997]. The benefits of buffer managers are well understood, particularly performance [Cai et al., 1997; Casas and Sevcik, 1989; Faloutsos et al., 1995; Garcia-Molina et al., 2000].

The set of buffers managed is known as the buffer pool, which is a resource acquired when the DBMS executes a query. Once a query is executed, another

³Refer to Hirzel [2007] and Lattner and Adve [2005] for efforts to provide fine-grained control of the data layouts of object-oriented programs.

query in the execution queue is selected and assigned one or more buffers based on the demand, current availability, and currently executing queries [Faloutsos et al., 1995]. Levy et al. [1996] reduces the problem of selecting a buffer for a query to a cache assignment problem—Given a truly random reference stream, only one buffer should be assigned to a given query.

The fundamental elements within each buffer are pages, which usually correspond directly to disk pages. When the DBMS requires access to a page not resident in the buffer, it must choose a page to evict to disk to accommodate the new page, called a page replacement (or eviction) policy. Once a page has been selected for eviction, either one of two cases can occur [Garcia-Molina et al., 2000]: (1) the evicted page has not been changed or (2) the evicted page has been changed. In the former case, the page is simply overwritten in the cache with the new page. In the latter, the contents of the page must be written back to disk. In some cases, the DBMS may subsume the buffers managed by the buffer manager using its own address space and implement a custom replacement policy [Effelsberg and Haerder, 1984].

Even though modern operating systems provide a similar mechanism by means of a file cache, most DBMSs do not leverage it [Effelsberg and Haerder, 1984; Stonebraker, 1981]. Conventional operating system replacement algorithms are not well suited for relational database environments [Stonebraker, 1981]. Most buffer managers in relational DBMSs control main memory directly, while object-oriented DBMSs and “main memory” databases often allocate buffers into virtual memory, delegating management to the operating system [Garcia-Molina et al., 2000]. Despite the disparity between the file cache and the buffer manager, most buffer managers divide the database into pages of the same size as those used by the underlying operating system [Effelsberg and Haerder, 1984].

Buffer managers typically select a fixed size to accommodate a static number of buffers [Effelsberg and Haerder, 1984]. This size should be limited so that it can fit in physical memory [Garcia-Molina et al., 2000]. The size may exceed physical memory and extend into virtual memory, however, this may lead to thrashing or the double paging phenomenon [Brice and Sherman, 1977; Chew and Silberschatz, 1992; Fernandez et al., 1978; Goldberg and Hassinger, 1974; Lang et al., 1977; Sherman and Brice, 1976a,b; Tuel, 1976].

2.2.1 Reference Behavior

Each page request to the buffer manager is a logical request. A series of such requests form a logical page reference string [Effelsberg and Haerder, 1984]. Each logical reference may or may not be associated with a physical reference, depending on the state of the buffers. Consequently, a logical page reference string can result in many different physical page references, even during the same run of the DBMS.

Understanding the patterns in the logical page reference string allow the buffer manager to take advantage of *locality*. Reference locality refers to the tendency of the DBMS to localize its work within a set of pages [Casas and Sevcik, 1989]. Chou and DeWitt [1985] observe three categories of reference patterns:

- **Sequential** reference patterns are characterized by references that occur one after another, with little or no repetition; some sequential references occur in a loop, referred to as a sequential looping pattern. In other cases, the sequential access backs up a few pages and continues (e.g. in a join) known as a clustered sequential pattern. All reference patterns not fitting into any other category are classified as clustered sequential.

- **Random** reference patterns access pages in statistically random sequences. Similar to sequential references, there can be a clustered random reference pattern.
- **Hierarchical** reference patterns access pages in a tree-like manner, from some root to leaves. Straight sequential refers to a single traversal of the tree structure while hierarchical with straight sequential refers to a reference pattern in which the tree traversal is followed by a sequential scan of the leaf nodes.

Definition 6. reference locality: *The natural tendency of a Database Management System to localize within a set of database pages.* [[Casas and Sevcik, 1989](#)]

Several systems attempt to predict what reference pattern will be required when a buffer is allocated, such as RESBAL [[Cai et al., 1997](#)], based on the resident set model. An understanding of a persistent application's reference behavior assists the developer or buffer manager in selecting the most appropriate page replacement policy.

2.2.2 Page Replacement Policies

When a page not resident in the buffer is accessed (page fault, or miss), a page must be evicted from the buffer to make room for the non-resident page. Page replacement policies can be categorized into the following approaches:

2.2.2.1 Basic

Basic page replacement strategies include naïve and static policies, those making predefined decisions based on primitive algorithms. The algorithms are simple to implement, but may not be well suited to all reference patterns. Because they are adaptations of common cache replacement algorithms within operating systems, they fail to account for access patterns common to the various DBMSs [Chou and DeWitt, 1985]. Additionally, the static nature of the basic algorithms mean they cannot adapt to changes in load or query behavior [Faloutsos et al., 1995].

Least-recently Used

The least-recently used (LRU) replacement strategy is a straight-forward algorithm which evicts the page which has not been used for the longest time. In order to facilitate determining what page was least-recently used, the buffer manager maintains an access table with a representation of the last access time per page. This strategy is effective for many reference patterns and its intuitive nature makes implementation simpler than many other policies [Garcia-Molina et al., 2000]. Numerous variations on this basic strategy have been proposed, such as LRU-k [O’Neil et al., 1993].

The clock algorithm is an efficient approximation of LRU, without less overhead of time tracking for each page [Garcia-Molina et al., 2000]. The pages are ordered logically in a circle, with markers identifying candidacy for replacement. A pointer points to one and only one page at any time. When a page is accessed, it is marked as an invalid candidate. When a page must be evicted, the buffer manager looks for the first page marked as a valid candidate by circulating clock-

wise from the current pointer. If it encounters a page marked not marked as evictable, it then marks it as such and moves the pointer again. Consequently, a page is evicted from the buffer as long as it has not been accessed in the time it takes the pointer to traverse the entire “clock” twice. Variations on the basic clock algorithm have been proposed, such as WS-clock [Carr and Hennessy, 1981] and Clock-Pro [Jiang et al., 2005].

Most-recently Used

Most-recently used (MRU), in contrast to LRU, evicts the most-recently used pages first. MRU therefore only requires a data member referencing the last page used, rather than a data member per page like LRU. However, MRU is not commonly used as few reference patterns match up well with the MRU behavior.

First-in First-out

First-in first-Out (FIFO) simply uses a queue to determine what page to evict. FIFO addresses the memory inefficiencies required to maintain the access time table in the LRU strategy. FIFO is often a rather ineffective strategy but is simple and easy to implement [Garcia-Molina et al., 2000].

2.2.2.2 Page-type Oriented

Page-type oriented replacement strategies include policies that classify the pages in the DBMS in order to more effectively isolate requests for pages of the same type. Although some claim increased throughput [Chou and DeWitt, 1985], [Effelsberg and Haerder, 1984] argues that page-type oriented schemes do not offer significant performance improvements over the basic strategies.

Domain Separation

Domain separation strategies categorizes pages into groups, or domains. Buffers are allocated by the buffer manager to a domain. Each of the buffers in each domain are managed using the LRU discipline. Although it can provide 8-10% improvement in throughput, the means by which it performs categorization of pages remains static [Chou and DeWitt, 1985]. The LRUI extension of the domain separation algorithm attempts to solve this problem by using priority rankings within domains [Chou and DeWitt, 1985].

“New” Algorithm

The “New” algorithm divides the buffers per relation in the database [Chou and DeWitt, 1985], in much the same way that the domain separation algorithm allocates the buffer pool to domains. Within each relation the MRU replacement policy is employed. This algorithm attempts to take advantage of *locality* within relations; however, when used in an experimental version of INGRES, the “New” algorithm failed to improve performance in any significant manner [Stonebraker, 1981]. The “New” algorithm primarily fails because it relies on the often ineffective MRU replacement within each relation.

2.2.2.3 Reference-pattern Oriented

Reference-pattern oriented replacement policies, unlike basic and page-type oriented replacement strategies, aim to maximize the number of page hits and minimize evictions by predicting or tailoring the buffer manager to the reference pattern of the application. Reference pattern oriented strategies may allocate more buffers to a query dynamically based on the expected reference pattern

[Faloutsos et al., 1995]. However, these strategies often impose heavy runtime performance penalties to perform prediction [Faloutsos et al., 1995], but usually outperform basic or page-type oriented schemes.

Hot-Set

Hot-Set is a predictive behavior model that is often used in relational databases to integrate advanced knowledge of the application’s reference pattern into the algorithm [Sacco and Schkolnick, 1982]. A hot-set refers to a set of pages used frequently, often within a sequential looping pattern. If the buffer allocated to the query is large enough to hold all the pages in the hot-set, the number of pages evicted will be minimized [Chou and DeWitt, 1985].

Definition 7. hot-set: *A set of frequently used pages in the query evaluation of a DBMS. [Sacco and Schkolnick, 1982]*

DBMIN

DBMIN is based on the query locality set model (QLSM) [Chou and DeWitt, 1985]. QLSM is predicated on the observation that under normal circumstances, DBMSs, particularly relational, exhibit common reference patterns in query evaluation. The locality set is the number of buffers required by the evaluation of a query to avoid inducing page faults [Faloutsos et al., 1995]. DBMIN attempts to allocate buffers such that they are as large or larger than the locality set. [Chou and DeWitt, 1985] claims DBMIN increases throughput 7-14% when compared to hot-set.

2.3 Java

This thesis focuses on the Java programming language. Java is a high-level, object-oriented, managed, general purpose programming language. Java code is compiled into an intermediate and platform independent bytecode representation for interpretation within a Java virtual machine (JVM). Java has increased in popularity over the past decade and is used as the implementation language in a growing number of applications [[Atkinson et al., 1996](#); [Rayside and Mendel, 2007](#)].

2.3.1 Object Model

As an object-oriented language, Java provides the facility for the construction of rich object models to encapsulate data and the operations on that data, including inheritance and abstraction. These models prescribe the type, cardinality, and direction of relationship between the objects. Instances of this model, sometimes referred to as a global object graph, contain nodes for each object created and a directed edge for a relationship between two objects [[Hirzel et al., 2002](#)].

The operations performed on an object depend on the runtime type of the object rather than its compile-time type [[Shaham et al., 2001](#)], providing polymorphic behavior. To facilitate this language feature, each object carries additional state in the form of an object header [[Bacon et al., 2002](#)]. To help identify the runtime type of an object, the Java virtual machine (JVM) attaches a run-time type identifier to the object. In most cases, this type identifier is a pointer to a type information block (TIB) in the virtual method table [[Shaham et al., 2001](#)]. The TIB holds information about a type and is used by the JVM for efficient

method invocation and dynamic type checking.

Additionally, there are three other common pieces of data placed in the header of each object. The default hash code provides the JVM assigned default hash code, usually some alteration of the object's memory address. Secondly, a lock is used to ease synchronization of objects, either with a lock representation defined directly in the header or a pointer to some lock object. Lastly, most JVMs use garbage collection techniques that require additional information to be placed in the header.

2.3.2 Reference Types

The Java programming language provides different types of references: strong, soft, weak, and phantom [Sun Microsystems, Inc., 2006b]. Hallett and Kfoury [2005] provides a survey of weak references in many languages, including Java. Strong references are the default reference type and establish a connection to an object that is considered during reachability analysis of garbage collectors. Weak references, on the other hand, are not used during reachability analysis and do not prevent their referent from being reclaimed. Soft references are similar to weak references, but leave their reclamation to the discretion of the garbage collector. Soft references remain until memory pressure (as identified by the virtual machine) causes their referent to be garbage collected. Phantom references are unlike weak and soft references in that they prohibit access to their referent [Agesen and Garthwaite, 2001].

Non-strong references have been used to implement property tables [Leal and Ierusalimsky, 2004], caches [Donnelly et al., 2006], weak sets, finalization [Leal and Ierusalimsky, 2004], object sampling [Agesen and Garthwaite, 2001]. Don-

nelly et al. [2006] developed λ_{weak} as formal calculus to describe the behavior of weak references. Gabay and Kfoury [2007] extended λ_{weak} with $\lambda_{multiref}$ as the foundation for a model of reachability-based garbage collectors. Despite their utility, non-strong references are rarely used because of their complex semantics [Donnelly et al., 2006].

2.3.3 Runtime Data Areas

2.3.3.1 Heap

Instantiation is the process of requesting memory be allocated for storing an object. Most objects instantiated within the JVM are allocated on the heap⁴. Such allocations may require the heap to grow in size to accommodate the new object. The heap can be viewed as a directed (possibly cyclic) graph with nodes for each object and references as edges [Hirzel et al., 2002]. Allocation consequently correlates to a node insertion.

During the initialization of the JVM, the heap is initialized with a maximum size. However, the JVM may not request that amount of memory at first from the operating system. Therefore, the maximum heap size represents a virtual memory size, not physical memory size (see §2.1.2). The swapping between physical and virtual memory for heaps that grow beyond their physical limits can introduce catastrophic performance degradations, commonly referred to as thrashing [Denning, 1970; Garcia-Molina et al., 2000]. By default, the heap will automatically grow or shrink in an effort to maintain the ratio of used to unused memory within a particular range [Sun Microsystems, Inc., 2006b]. However, depending on the allocation algorithm, this ratio varies significantly. Many ap-

⁴Some objects may be allocated directly to the stack, using techniques such as escape analysis [Hirzel et al., 2002].

plications, such as large-scale servers, usually must tune these growth parameters for optimal performance [Sun Microsystems, Inc., 2006b].

2.3.3.2 Stack

The stack is a structure for holding temporary data and intermediate results. The stack stores one or more frames, which stores data and partial results. Each method invocation causes a new frame to be created on the stack and likewise deleted when the method returns. Each frame is subdivided into three regions.

Local Variables

The local variable region of a frame stores local variables for the frame indexed starting at zero. Each local variable index position is occupied by a word, so values of type `long` and `double` occupy two indices in the local variable region (as they require two words to be represented). Values are passed to methods using the local variables region. When an instance method is being invoked, the local variable at index 0 is implicitly a reference to the `this` instance, and subsequent indices are local variables passed to the method. When a static method is invoked on a class, the local variables are indexed starting at 0. The size of the local variables index is specified at compile time.

Operand Stack

Each frame contains an operand stack on which instructions operate. The operand stack can hold any value in a single entry, while `long` and `double` values increase the depth by two. All operations performed on the operand stack must be type-safe. The size of the operand stack is specified at compile time.

Runtime Constant Pool

To support dynamic linking, each frame references the runtime constant pool. Method invocation and variable access are accomplished using symbolic references, which are dynamically translated into concrete references using the constant pool (in support of JIT compilation). Each class file maintains a runtime constant pool to store constants, method references and fields references. For example, in Listing 21 of Appendix C, line 19 loads a string constant and line 5 resolves an interface types from the class' constant pool.

2.3.4 Java Object Serialization

Java Object Serialization (JOS) has been supported in the Java since version 1.1 and provides a core capability to read and write Java objects to and from raw byte streams. JOS is integral to the Remote Method Invocation (RMI) component and integrated within JavaBeans. In order for an object to acquire the capability to serialize itself, all it must do is implement the `Serializable` interface, which has no methods⁵. All non-static, non-transient data, including primitives, is automatically serialized when requested [Kurotsuchi, 1997]. Should the serialization request require the serialization of an object not marked with the `Serializable` interface, the serialization will fail. The resulting byte stream can be deserialized into an object of the same type that was serialized.

JOS can be used natively as a primitive form of persistence [Ortín et al., 2004]. However, on its own it is often considered unsuitable to large scale persistence environments and is often wrapped by a database management system. While

⁵The `Serializable` interface provides no public methods, but allows implementations to override the `readObject` and `writeObject` methods by convention. On the other hand, classes can implement the `Externalizable` interface and explicitly provide implementations of the `readExternal` and `writeExternal` methods.

commonly used for file-based persistence, Java object serialization can be used to store binary data into a relational (using JDBC) or even a non-relational database [Olson et al., 1999]. A shortcoming of JOS as a persistence methodology is the lack of serialization support for static variables [Jordan and Atkinson, 2000].

2.3.5 Bytecode

Bytecode is typically stored in a class file (.class), which represents a single class or interface as an octet stream. Listing 21 in Appendix C shows an example of the bytecode constructed for a instance method of a class. Types are represented using descriptors, which is the union of the qualified name of a type and a field type character from Table 1 [Lindholm and Yellin, 1999]. For example, a method that takes a single integer argument and returns an object has the descriptor `<(I)Ljava/lang/Object;>`.

Table 1: Descriptor types

B	byte	signed byte
C	char	Unicode character
D	double	double-precision floating point
F	float	single-precision floating point
I	int	integer
J	long	long integer
L< class >;	reference	reference to < class > instance
S	short	signed short
Z	boolean	boolean
[	reference	one array dimension

Each instruction is represented by a single byte opcode and optionally one or more parameters. Each operation is stack-oriented, thus requiring all input and output to be present on the stack. A complete listing of the supported bytecode instructions can be found in Appendix D.

2.3.5.1 Bytecode Engineering

There exist several tools to perform bytecode engineering as a post-compilation or runtime process. The most popular tools include the Bytecode Engineering Library (BCEL) [[Apache Software Foundation, 2006](#)], Javassist [[Chiba, 2000](#)], and ASM [[ObjectWeb, 2007](#)]. These tools allow for inspection, mutation or generation of bytecode in its native form. While typically aimed at monitoring or profiling [[Binder et al., 2007](#); [Pearce et al., 2007](#)], bytecode manipulation is a commonly used technique to solve a variety of problems [[Binder et al., 2007](#); [Choi, 2000](#); [Garcia et al., 2005](#); [Kiczales et al., 2001](#); [Richardson and Schwarz, 1991](#); [Tanter et al., 2002](#); [Vallée-Rai et al., 1999](#); [Villazón et al., 2008](#)]⁶. While the Java programming language does offer reflection capabilities, it does not support altering the program semantics at runtime. JDK 1.6 provides instrumentation agents which can allow instrumentation of classes while loading [[Sun Microsystems, Inc., 2006b](#)]. However, this feature is rather limiting in that it fails to support instrumentation of fields [[Binder et al., 2007](#)]. Bytecode engineering provides a suitable alternative for post-compilation semantics alteration.

ASM

ASM is a popular bytecode engineering library that can be used to generate, transform, and analyze Java classes [[Bruneton, 2007](#)]. ASM objectifies the contents of a class file and provides two alternative parsing implementations: (1) an event-based stream parser and (2) an object-based tree parser. ASM transformation occurs statically (compile-time) rather than dynamically.

⁶There is an increasing trend to use bytecode manipulation as a means for aspect-oriented programming [[Elrad et al., 2001](#)] (see §2.4).

2.4 Aspect-oriented Programming

Object-oriented programming, now considered a mainstream software development approach, has several limitations. Most significantly, some functionality does not decompose well into single units [Elrad et al., 2001]. These functional elements are known as cross-cutting concerns. A concern embodies a concept, goal, or functional unit [Kiczales et al., 2001]. Common examples of cross-cutting concerns include synchronization, debugging, security, resource sharing, distribution, and memory management [Rashid and Loughran, 2003; Rashid and Pulvermuller, 2000]. Although some of these concerns can be encapsulated using design patterns, many are not suitable or complete solutions [Rashid and Pulvermuller, 2000].

Object-oriented programming is subject to lack of *functional locality* [Filman and Friedman, 2005], a situation in which the functionality is spread across multiple levels of inheritance. An understanding of the classes being inherited is vital for effective and safe development, as illustrated by the fragile base class problem [Mikhajlov and Sekerinski, 1998]. Kiczales et al. [2001] argues that the hierarchical modularity mechanisms of object-oriented languages are unable to modularize all concerns in complex systems. Rather than conforming a solution using the structures of the language, it should be possible to specify modularity the way programmers naturally think about the system. Elrad et al. [2001] argues that software is more appropriately engineered by specification of concerns separately, with some relation between them.

Aspect-oriented programming is a new approach to software development that aims to modularize cross-cutting concerns [Kienzle and Guerraoui, 2002; Rashid and Chitchyan, 2003]. As opposed to object-oriented programming, aspect-

oriented programming gathers scattered concerns spread through the system into a single location [Elrad et al., 2001]. These modularized concerns, called aspects, are promoted to first-class citizens [Kienzle and Guerraoui, 2002]. Modularizing a concern into one location localizes maintenance, reduces development costs and enhances readability [Kienzle and Guerraoui, 2002; Rashid and Loughran, 2003].

Aspect-oriented programming is not a replacement for object-oriented programming, but rather a complementary approach [Elrad et al., 2001]. Kiczales et al. [2001] argues that aspect-oriented programming does for cross-cutting concerns what object-oriented programming does for encapsulation. Despite this complementary relationship, aspect-oriented programming can be facilitated without object-oriented techniques [Filman and Friedman, 2005]. For example, a language supporting mixins and multiple inheritance meets aspect-oriented principles. Aspect-oriented programming is similar to adaptive programming [Kiczales et al., 2001] and generative programming [Katz, 1990].

Aspects agglomerate code that reside in disparate locations into one location [Rashid and Pulvermueller, 2000]. Aspects declaratively specify [Filman and Friedman, 2005]: *In a program P , whenever condition C arises, perform action A .* The condition, C , on which an aspect operates is called a pointcut. The actions applied whenever the pointcut holds true are called advice. The locations in the code that may take part in a pointcut are labelled join points [Kienzle and Guerraoui, 2002].

Aspects can have various levels of coupling with the entities they interact with, or are being modified by the aspect⁷. These levels of coupling are [Rashid and Pulvermueller, 2000]:

⁷Typically these entities are classes in an object-oriented language, but here they are referred to an “advisee” to disambiguate cases in which aspect-oriented programming is not paired with an object-oriented language.

- **Open** coupling occurs when the advisee and aspect have visibility to one another.
- **Class-directional** coupling occurs when the aspect has visibility to the advisee, not vice versa.
- **Aspect-directional** coupling occurs when the advisee has visibility to the aspect, not vice versa.
- **Closed** coupling occurs when neither the aspect nor the advisee has visibility to one another.

Aspect-oriented programming is being employed in the following areas of software engineering:

- Application server development [[Choi, 2000](#)]
- Orthogonal persistence [[Hohenstein et al., 2007](#); [Pulvermueller et al., 1999](#); [Rashid, 1998](#); [Rashid and Pulvermueller, 2000](#); [Rashid and Sawyer, 2000](#); [Rashid et al., 2000](#)]
- Transaction management [[Kienzle and Guerraoui, 2002](#); [Soares et al., 2002](#)]
- Distributed programming [[Pawlak et al., 2004](#); [Soares et al., 2002](#); [Tatsubori, 2001](#)]
- Inheritance and mix-ins [[Richardson and Schwarz, 1991](#)]
- Memory management [[Chen and Chen, 2007](#); [Schoeberl and Vitek, 2007](#)]
- Failure management [[Filho et al., 2006](#); [Kienzle and Guerraoui, 2002](#); [Lipert and Lopes, 2000](#)]

- Concurrency management [[Kienzle and Guerraoui, 2002](#)]
- Design patterns [[Cacho et al., 2006](#); [Gamma et al., 1995](#); [Hannemann and Kiczales, 2002](#)]
- Unit testing [[Kulesza et al., 2005](#)]

2.4.1 Weaving

A critical process within several aspect-oriented implementations is weaving. Weaving is the process of redistributing advice from aspects to the cross-cutting locations (join points) in the code the aspect is encapsulating. The advice in the aspect is woven into appropriate locations based on the join points. Aspect weavers translate aspect definitions and merge advice with their respective join point(s) [[Rashid, 2002](#); [Rashid and Pulvermuller, 2000](#)]. Weaving can be accomplished statically at compile-time or dynamically at run-time [[Rashid, 2002](#)], referred to as “load-time weaving,” or LTW. Static weaving is usually implemented as a pre- or post-compilation step (see §2.3.5.1), while load-time weaving is primarily accomplished by extending the run-time behavior of the code interpreter.

2.4.2 Discussion

The benefits of aspect-oriented programming are well documented [[Cacho et al., 2006](#); [Colyer and Clement, 2004](#); [Filho et al., 2006](#); [Garcia et al., 2005](#); [Hannemann and Kiczales, 2002](#); [Papapetrou and Papadopoulos, 2004](#); [Singh and Kiczales, 2007](#)]. These studies, in particular [Singh and Kiczales \[2007\]](#), confirm the applicability of aspect-oriented programming in large systems with minimal

impact on performance while improving readability, abstraction, and reducing coupling.

However, some disadvantages have hindered their widespread use. Aspect-oriented programming should not be applied to the singleton design pattern [Filman and Friedman, 2005]. No benefit is gained from applying an aspect-oriented paradigm to a concern that does not bridge multiple locations in the source code. Steimann [2006] argues that aspect-oriented programming fails to achieve goals of enhanced modularity and understandability. Due to the cross-cutting nature of advice within an aspect, which can span several locations in the source code, developers require visual feedback in order to assess the affect an aspect has across the various components [Kienzle and Guerraoui, 2002]. Consequently, it is generally understood that aspect-oriented programming requires effective development environment support⁸.

While some argue that “obliviousness” is paramount to aspect-oriented programming [Filman and Friedman [2005]], others caution against this approach. In contrast, Soares et al. [2002] suggest that the goal of aspect-oriented programming is not to make cross-cutting concerns completely transparent, but rather localized. The developer should be explicitly aware of the aspects applying to a particular join point. However, this can lead to *open coupling* and may limit reusability. Rashid and Loughran [2003] attempt to solve this problem by supporting the mining of aspects. Soares et al. [2002] suggests that due to the flexibility, power, and expressiveness of aspect-oriented programming, it should be used cautiously, as a slight change in an aspect can have undesirable effects on system.

⁸Lack of development environment support in early iterations of many aspect-oriented implementations prohibited large scale or commercial use. Recent development environment tools and extensions have mitigated this concern.

3 Problem

Memory is a constrained resource and undoubtedly will remain so despite advances in hardware. Systems running in these memory-constrained environments are referred to as memory-constrained systems. Such systems can consume much more memory than is available on general-purpose modern computing devices, including large-scale servers and embedded/mobile systems. Examples of memory-constrained systems include:

- Persistent Application Systems (PAS) [Atkinson and Morrison, 1995]
- Embedded systems [Panda et al., 2001]
- External memory (EM) algorithms [Ajwani et al., 2006; Arge, 1997; Chiang et al., 1995; Vitter, 1998, 2001]
- Expert systems [Marney and Ibrahim, 1995]
- Puzzle solving [Kunkle and Cooperman, 2008]
- Scientific computing [Zhang et al., 2005]
- Data mining [Kantardzic, 2002; Liu, 2008]
- Geographic information systems (GIS) [Arge, 1997; Haverkort et al., 2009]
- Ad-hoc sensor networks [Liu et al., 2005]
- XML processing
- Web servers

Definition 8. memory footprint: *The maximum amount of memory begin used at a given moment in time by a system.*

Definition 9. working set: *The set of data a system can operate on within the memory allotted to it.*

Definition 10. memory-constrained system: *A system bounded by limited memory resources whose working set is larger than the main memory it has been allocated.*

Definition 11. footprint reduction: *Process of reducing the memory footprint required by a memory-constrained system.*

Information-rich systems, including as ontologically based [Devedzić, 2002], expert and knowledge-based systems [Marney and Ibrahim, 1995], are constructed to represent a rich model of the real world. Doing so allows the the system to reason about the complex structure, relationships, and rules associated with entities of the model. The richness of the model comes at the cost of main memory. Similarly, computer-aided design (CAD), computer-aided engineering (CAE), computer-aided manufacturing (CAM), and computer-aided software engineering (CASE) tools [Atkinson and Morrison, 1995; Brown and Morrison, 1992; Kemper and Kossmann, 1995; Lamb et al., 1991] often operate on complex data structures. These systems are typically unnatural candidates for common persistence techniques, such as databases, yet they operate on large data sets.

The maximum data set that a memory-constrained system requires to correctly operate is referred to as its working set¹. Memory-constrained systems are those bounded by limited memory resources whose working set is larger than the main memory it has been allocated. As such, these systems must take effort to perform operations on their working set with precise control over memory management.

¹Adapted from Denning [1968].

Virtual memory management within the Java virtual machine lacks the fine-grained control required by memory-constrained systems. As such, the common solution taken by these systems is to push data into lower memory tiers explicitly. Memory-constrained systems typically overcome their memory limitations with custom memory management and I/O reduction techniques, such as caching. What separates these solutions is when and how often this process occurs. While doing so alleviates main memory pressure, it adversely affects performance, due to the increased amount of I/O incurred between adjacent tiers (refer to §2.1.1). However, explicit memory management often bounds the implementation to a specific problem space and limits reusability in other problem domains. In the opinion of the author, a consolidated framework supporting the needs of memory-constrained systems in a generic manner has yet to be constructed and validated.

Problem Statement. A generic, portable, and extensible framework supporting the configurable memory footprint reduction of memory-constrained systems in Java does not exist.

4 Solution

JDiet is proposed as a solution for footprint reduction of memory-constrained systems written in the Java programming language. Utilizing well-known memory management techniques, the goal of JDiet is to allow a memory-constrained system to increase the size of its working set by consuming less physical memory than is normally required. As the amount of memory consumed by heap data far outweighs the code, stack, and other runtime data areas [Choi and Han, 2008], a generic memory management solution must address heap data usage. In much the same way as virtual memory, JDiet provides the illusion of a much larger heap¹. However, JDiet is configured so that decisions about the entities available for eviction, and policies for eviction, are made at the behest of the system designer, not the virtual memory manager. Doing so allows the system designer to encode application-specific knowledge, such as reference behavior, into the memory management system. JDiet aims to support memory-constrained systems willing to sacrifice performance for a larger working set.

JDiet bases its approach on two intuitive observations: (1) *temporal cruciality* and (2) *intra-object locality*. The access time required to retrieve data within an object need not be constant. In many situations, slower access time to some parts of an object is tolerable. *Temporal cruciality* refers to the disparity between the desired access speeds for the attributes of an object. Those attributes not requiring high-speed access can be swapped from main memory to auxiliary memory to increase the size of the working set, at the cost of increased access time.

Definition 12. temporal cruciality: *The tendency for the access times of attributes of an object to have dissimilar degrees of importance of access time.*

¹Unlike virtual memory, JDiet makes no claims as to the virtual infiniteness of memory gained; rather, JDiet increases the apparent heap size by some finite amount.

For those parts of an object deemed least time critical, access is subject to *intra-object locality*. *Locality* is a well-known phenomena in databases [Effelsberg and Haerder, 1984; Lamb et al., 1991; Sacco and Schkolnick, 1986], memory management [Bilardi and Peserico, 2001; Denning, 1968, 1970; Grunwald et al., 1993], garbage collection [Alonso and Appel, 1990; Courts, 1988; Huang et al., 2004; Siegart and Hirzel, 2006; Wilson et al., 1991] and caching [Smith, 1982]. *Locality* is the tendency of an application to use only a partial subset of its data set at any given time. Locality is similar to the concept of “hot-regions” [Shuf et al., 2002]. JDiet hypothesizes that *locality* is not limited to objects, but can be applied to lower levels of granularity within each object; each object is subject to *locality* with respect to its attributes. Some attributes will inherently be accessed much more frequently than others. *Temporal cruciality* and *intra-object locality* are correlated, as the subset of attributes of an object accessed more frequently intuitively require faster access times.

Definition 13. intra-object locality: *The tendency of an application to use only a partial subset of an object’s attributes at any given time.*

Theorem 1. intra-object locality implies cruciality: *The locality of access within an object implies the cruciality of the access times of those attributes accessed most frequently.*

Those attributes which are accessed more frequently are referred to as prime attributes, and those accessed less frequently referred to as non-prime attributes. JDiet allows the developer to identify the non-prime attributes of a class as *fat attributes*. Subsequently, JDiet extracts these *fat attributes* from a class and forms a new *fat class*². The source class from which the non-prime attributes are

²This approach is novel in that it requires no pointer swizzling. *Fat attributes* may only of primitive data type, therefore removing the requirement to persist pointers to objects. The process of swizzling can be costly and significantly affect the complexity of the solution.

extracted is subsequently referred to as a *thin class*.

Definition 14. thin class: *A class whose non-prime attributes have been extracted to a fat class.*

Definition 15. thin object: *An instance of a thin class.*

Definition 16. fat class: *A class that stores non-prime attributes extracted from a thin class.*

Definition 17. fat object: *An instance of a fat class.*

Definition 18. thinning: *The process of extracting non-prime attributes from an input class into a fat class, resulting in a thin class.*

JDiet currently presumes that primitive data is the primary subject of memory consumption³. Only non-static fields with primitive data types, including `Strings` and `Arrays` are supported *fat attributes*. Reference types are not considered *fat attributes*, as references do not pose considerable memory strain⁴, but rather the primitive data contained in the objects they reference. Primitive data, `Strings` in particular, are the cause of significant memory usage for normal Java applications. For example, [Kawachiya et al. \[2008\]](#) cites that approximately 40% of objects in the heap for a J2EE application server are `String` objects.

4.1 Design Principles

The design of JDiet is subject to the following architectural principles. These principles ensure JDiet is engineered for modularity, minimization of cross-cutting concerns, and memory reduction, while providing qualitative metrics to contrast JDiet with alternative solutions.

³See §7.1 for further discussion

⁴In Java, each reference consumes a single word, which is commonly 4 or 8 bytes.

*Principle 1. **Memory-driven.*** In situations where a tradeoff between memory and performance must be made, performance should be sacrificed in favor of memory.

*Principle 2. **Minimal overhead.*** The overhead of the memory management system should be significantly less than the memory gained by using the system.

*Principle 3. **Semantic preservation.*** The behavior of the application should not be altered by the existence of the memory management system.

*Principle 4. **Configurability.*** The system designer is ultimately responsible for making informed engineering decisions as to what entities of the memory-constrained system can be subject to management.

*Principle 5. **Extensibility.*** The system designer should be capable of customizing or extending the behavior of the memory management system.

*Principle 6. **Transparency.*** There should exist no explicit compile-time dependency between the entities being managed and the management system itself.

*Principle 7. **Type Orthogonality.*** The types applicable for management are orthogonal to the modeling decisions made by the system designer.

*Principle 8. **Portability.*** The system should run natively within the Java HotSpot virtual machine, requiring no changes to the language or the language interpreter.

4.2 Approaches

Several techniques can be used to solve the problem identified in §3, while their adherence to the design principles vary. Common to each technique is the ability to identify the classes subject to management and intercept access to the *fat attributes* of those classes. Four technical approaches can be employed: (1)

virtual machine alteration (2) reflection (3) bytecode engineering and (4) source alteration. The tradeoffs of each approach, with respect to the guiding principles, are outlined below.

4.2.1 Virtual machine alteration

This technique focuses on changes to the underlying behavior of the Java virtual machine implementation. At this level, all features of the JVM can be exploited, such as memory allocation, reference counting, and finalization. The most obvious technique is to add a new reference type to the core language, one that establishes semantics for a reference to data not guaranteed to be in main memory. The residency of the referenced data can be detected by the virtual machine and swaps handled accordingly. Such an approach is similar to that taken by persistent programming languages [Atkinson and Morrison, 1995] and proposed as “virtual memory primitives” [Appel and Li, 1991]

Despite the benefits of this approach, virtual machine alteration limits the applicability of the system to widespread use. This technique is applicable in embedded or mobile computing environments but limiting elsewhere. Unless alterations are made to the Java HotSpot virtual machine, such an approach is unlikely to gain widespread adoption. While custom virtual machines are employed in various circumstances, taking this approach violates the principle of *portability*.

4.2.2 Reflection

This technique uses the built-in reflection capabilities of Java to instrument classes. While this technique alleviates concerns of an alternative JVM imple-

mentation, reflection has limited functionality and performance impact. The dynamic proxy mechanism in Java is limited to the interception of method calls, specifically those in the dynamic proxy interface [Sun Microsystems, Inc., 2006b]. Consequently, this approach requires coding conventions, such as plain old Java objects (POJOs) or JavaBeans, to expose the attributes of a class through a public interface. While this may not be a limiting factor for the construction of new memory-constrained systems, it limits the applicability to existing systems which do not conform to either of those conventions. This violates the guiding principle of *transparency*. Furthermore, Java’s reflection mechanism is discouraged for performance-sensitive applications [Sun Microsystems, Inc., 2006b] due to the inherent performance cost of reflective access. Techniques are being explored to offset the traditional costs of reflection, such as partial evaluation [Braux and Noyé, 1999], but these techniques are not widely utilized.

4.2.3 Bytecode engineering

This technique uses bytecode manipulation techniques to alter or construct entities to intercept attribute access. Bytecode engineering physically modifies compiled class files in a post-compilation process. This approach can optimize performance roughly equivalent to virtual machine modification, but does require manipulation of code. This manipulation requires an additional build step, but is gaining acceptance in the aspect-oriented [Elrad et al., 2001; Kiczales et al., 2001] and generative programming [Tanter et al., 2002] communities. Bytecode engineering separate the concerns of memory management from the source code, requiring no alteration to the source.

4.2.4 Source alteration

This approach is the most primitive and explicit. Unlike the previous approaches, source alteration comprises the various techniques for explicit modification of source code to perform memory reduction. This technique employs a persistence technology to store object data to auxiliary memory and manually retrieve the objects again before their use. Most persistence technologies require the system designer to explicitly specify the persistence approach at compile-time [Al-Mansari et al., 2007]. This includes identifying the persistent types, performing any type adaptation or mapping to the persistence model, and making explicit calls to the persistence application programming interface. This process is contrary to the principles of *transparency* and *type orthogonality*. While straightforward, this approach is limited in that eviction decisions are hard-coded into the source by the system designer (at a micro level). Doing so limits the ability of the memory management system to optimize eviction based on the runtime reference behavior (at a macro level).

4.2.5 Selected Approach

JDiet takes the bytecode engineering approach, as it aligns best with the guiding principles. The most significant benefit of bytecode engineering is *portability*. Modification to binary class files requires no changes to the original source, but allows the memory management behavior to be overlaid on existing compiled source. The secondary benefit of binary modification is separation of memory management from the source code as a cross-cutting concern.

While virtual machine alteration is a valid and commonly used technique in academia [Blackburn et al., 2006; Bond and McKinley, 2008; Hertz and Berger,

2005; Jump and McKinley, 2007; Tang et al., 2008; Zhang and Krintz, 2005], particularly the use of the Jikes Research Virtual Machine [Jike RVM Project, 2009], doing so would hinder the applicability to the various operational environments it aims to support. Virtual machine modification and bytecode engineering both provide *transparency* and *type orthogonality*, but virtual machine modification lacks extensibility and configurability; unless aimed at a specific software product, a virtual machine with custom memory management is unlikely to support customization or configuration.

4.3 Architecture

JDiet is composed of two subsystems, the Compile-time Weaver (CTW) and Runtime Manager (RM), as illustrated in Figure 2. The CTW is responsible for the identification, classification, and interface implementation required by the RM. The Runtime Manager manages *fat objects* and is responsible for providing access to those instances.

4.3.1 Compile-time Weaver

As its name implies, the CTW performs compile-time bytecode engineering. JDiet is architected to physically extract *fat attributes* from classes. Guided by a configuration (as illustrated in Figure 3), the CTW identifies the classes and attributes subject to *thinning*. Unlike persistence preparation techniques identified by Al-Mansari et al. [2007], this approach decouples the identification of managed entities from the entities themselves, similar to well-known aspect-oriented techniques. With respect to aspect-oriented programming, the CTW employs *class-directional coupling* (discussed in §2.4). Those classes targeted in

Figure 2: JDiet subsystems

the configuration are subsequently *thinned* by extracting a *fat class* to store the *fat attributes*.

Figure 3: Compile-time Weaver component architecture

The algorithm for the fat extraction process is described by Algorithm 1. The CTW takes as input a set of classes exposed to the weaver, C_θ , and a configuration, F ; both must be non-empty. The classes in C_θ must be compiled class files (ending with the `.class` extension) and exposed by F . Not all classes exposed to the weaver are subject to thinning. Lines 1 through 2 initialize the data structures used to store the *thinned* and *fat classes*, C_Δ and C_F respectively. A double nested loop is used to iterate over all the class configurations within F (lines 3 to 13). For each class specified in the configuration, F_c , a new *fat class* is constructed, C_f . The class for which one or more attributes to be extracted is assigned to a local variable, C_δ on line 5. Lines 6 to 10 perform the extraction of each attribute, A , specified by the configuration, F_a in F_c . The attribute is extracted from the *thin class* (line 8) and placed into the *fat class* (line 9). Once the *thinning* of a class is complete, the *thin* and *fat classes* are appended to the running sets. Once all class configurations, F_c , have been executed, the algorithm returns the set of modified (thinned) input class, C_Δ , and associated *fat classes*,

C_F . The quantity of *fat classes* and *thin classes* must be equal after completion, but may be greater than the number of classes configured for *thinning*. The union of the *thin class* and the *fat class* are semantically equivalent to the original input class, according to Theorem 2, to adhere to the principle of *semantic preservation*.

Algorithm 1 CTW fat extraction

Require: $C_\theta \neg \emptyset$

Require: $F \neg \emptyset$

```

1:  $C_\Delta \leftarrow \emptyset$ 
2:  $C_F \leftarrow \emptyset$ 
3: for all  $F_c \in F$  do
4:    $C_f \leftarrow \emptyset$ 
5:    $C_\delta \leftarrow F_c \in C_\theta$ 
6:   for all  $F_a \in F_c$  do
7:      $A \leftarrow F_a \in C_\delta$ 
8:      $C_\delta \leftarrow C_\delta - \{A\}$ 
9:      $C_f \leftarrow C_f \cup A$ 
10:  end for
11:   $C_\Delta \leftarrow C_\Delta \cup C_\delta$ 
12:   $C_F \leftarrow C_F \cup C_f$ 
13: end for
14: return  $C_\Delta, C_F$ 
Ensure:  $|C_\Delta| = |C_F| \geq |F|$ 

```

Theorem 2. Semantic Equivalence: For every class configured for thinning, there exists a pair of classes $\langle C_f, C_\delta \rangle$ returned from Algorithm 1 such that the union of their contents is semantically equivalent to the original input class.

$$\forall F_{c_i} \in C_\theta : \exists (C_{f_i} \in C_F, C_{\delta_i} \in C_\Delta) \ni C_{\theta_i} \equiv C_{f_i} \cup C_{\delta_i} \quad (4.1)$$

4.3.2 Runtime Manager

The *fat classes* extracted by the compile-time weaver are subsequently managed by the Runtime Manager (RM). *Thin objects* access their associated *fat objects* through the RM. As illustrated in Figure 4, the RM provides two func-

tional capabilities, manage and load.

Figure 4: Runtime Manager interaction with thin and fat classes

As instances of *thin classes* are constructed they instantiate an instance of their associated *fat class* and register the *fat object* instance with the RM by invoking the **manage** interface. The *thin object* maintains a logical reference to the *fat object*, but accesses it physically by dereferencing through the **load** interface of the RM. The RM is composed of the four modular components shown in Figure 5: (1) buffer manager (2) buffer (3) eviction policy and (4) store.

4.3.2.1 Buffer Manager

The **BufferManager** implements the **manage** and **load** interfaces, described in Algorithms 2 and 3. The **BufferManager** relies on an underlying **Buffer** to manage a discrete set of *fat objects*. The **BufferManager** provides static singleton access to a **Buffer** instance. Both the **manage** and **load** operations are delegated to the **Buffer**. While the **load** operation is a direct pass-through, the **BufferManager** is responsible for identity assignment (lines 1 - 2 of Algorithm

Figure 5: Runtime Manager component architecture

2) before passing the *fat object* to the **find** method of the **Buffer** instance. The identity assigned to the fat object is returned by the **manage** function so that the *thin object* making the call to **manage** can reference the *fat object* logically. Subsequent access to the *fat object* is made by calling the **load** method and passing the identity.

Algorithm 2 Buffer Manager: **manage**

Require: O_f

- 1: $id_f \leftarrow \text{genID}()$
 - 2: $id(O_f) \leftarrow id_f$
 - 3: **Buffer.insert**(O_f)
 - 4: **return** id_f
-

Algorithm 3 Buffer Manager: **load**

Require: id_f

- 1: **return** **Buffer.find**(id_f)
-

4.3.2.2 Buffer

The **Buffer** is the component that manages all the *fat objects* resident in main memory. Each **Buffer** is required to facilitate the insertion and retrieval of *fat objects* through the **insert** and **find** interfaces respectively. Each **Buffer** is associated with a **Store**, which manages *fat objects* that cannot fit in the **Buffer** and must be evicted to auxiliary memory. Consequently, each *fat object* instance can only reside in either the **Buffer** or **Store** at any given moment, as stated by the Theorem of *monolocation*.

Theorem 3. *Monolocation: A fat object instance, O_f , exists in either the **Buffer**, B , or the **Store**, S , but not both, at any moment in time.*

$$O_f \in (B \cup S) \wedge B \cap S = \emptyset \quad (4.2)$$

Insert

The algorithm for the `insert` method is provided in Algorithm 4. The insertion process requires a new *fat object*, O_f and the size of the `Buffer`, $\hat{B}$ (which is configured by the system designer). An error check is performed from lines 1 to 5 to ensure the *fat object* being inserted does not already exist in the `Buffer`. The crux of the algorithm begins on line 6. If the `Buffer` is full, a *fat object* must be evicted to the underlying `Store` to make room for the *fat object* being inserted. This process is referred to as an *object swap* and is implemented between lines 6 and 10. The `Buffer` delegates the responsibility of selecting the *fat object* to evict to the `EvictionPolicy`, through the implementation of the `select` interface method. Once the *fat object* to evict has been selected, O_δ , it is pushed to auxiliary memory through the `put` method of the `Store`. Once moved to the `Store`, the *fat object* evicted is removed from the `Buffer`. The insertion ends by placing the new *fat object* into the `Buffer` at line 11. This process ensures the Theorem of *monolocation* is upheld⁵.

Definition 19. object swap: *The physical exchange of two objects, one resident in main memory and the other resident in auxiliary memory.*

Find

The `find` method of the `Buffer` facilitates access to *fat object* instances inserted through calls to the `insert` method. The algorithm for the `find` method is provided in Algorithm 5. This identifier is the logical reference stored in the *thin object* to the *fat object* (created during the `manage` method in the

⁵Technically speaking there is a point in time within the execution of the `insert` method during which the *fat object* is neither in the `Buffer` nor the `Store`; however, *monolocation* is upheld by the time the `insert` method returns.

Algorithm 4 Buffer: insert

Require: O_f **Require:** $\hat{B}$

```
1: for all  $B_i \in B$  do
2:   if  $id(B_i) = id(O_f)$  then
3:     return error
4:   end if
5: end for
6: if  $|B| = \hat{B}$  then
7:    $O_\delta \leftarrow \text{EvictionPolicy.select}(B)$ 
8:    $\text{Store.put}(O_\delta)$ 
9:    $B \leftarrow B - \{O_\delta\}$ 
10: end if
11:  $B \leftarrow B \cup \{O_f\}$ 
```

BufferManager). The `find` takes as input the identity of the *fat object* to be searched for. Additionally, the algorithm requires a maximum buffer size be known in advance, $\hat{B}$. The algorithm can be divided into two sections: (1) buffer hit and (2) buffer miss. Lines 1 - 5 search iteratively through identities of the current *fat objects* in the **Buffer** to determine if a buffer hit has occurred. Lines 6 - 13 handle a buffer miss by retrieving the desired *fat object* from the underlying **Store**. The process of retrieving *fat objects* from auxiliary memory not resident in main memory is known as an *object fault*. Either one of two states can be encountered when a buffer miss occurs: (1) the buffer is full (2) the buffer is not full. If the buffer is full, a *fat object* must be evicted. This process follows the same algorithm as that taken in Algorithm 4. A *fat object* in the **Buffer** is selected using the `EvictionPolicy` and is pushed to the **Store**. Once sufficient room has been made, the *fat object* that was faulted is pulled from the **Store** with a call to the `remove` method (line 11). The *fat object* instance retrieved is re-inserted into the **Buffer** and returned.

Definition 20. buffer hit: *Situation in which a fat object being accessed is resident in the buffer.*

Definition 21. buffer miss: *Situation in which a fat object being accessed is not resident in the buffer.*

Definition 22. object fault: *Buffer miss requiring an object swap.*

Algorithm 5 Buffer: find

Require: id_f

Require: $\hat{B}$

```

1: for all  $B_i \in B$  do
2:   if  $id(B_i) = id_f$  then
3:     return  $B_i$ 
4:   end if
5: end for
6: if  $|B| = \hat{B}$  then
7:    $O_\delta \leftarrow \text{EvictionPolicy.select}(B)$ 
8:    $\text{Store.put}(O_\delta)$ 
9:    $B \leftarrow B - \{O_\delta\}$ 
10: end if
11:  $O_f \leftarrow \text{Store.remove}(id_f)$ 
12:  $B \leftarrow B \cup \{O_f\}$ 
13: return  $B_\delta$ 

```

4.3.2.3 Eviction Policy

The **Buffer** component relies on the implementation of the **EvictionPolicy**. The **EvictionPolicy** is responsible for choosing the next suitable *fat object* for eviction from the **Buffer** by implementing the **select** method. The **select** method implementation is required to remove the *fat object* from management or storage in the **EvictionPolicy**, if any is required. The **insert** and **touch** methods allow the **EvictionPolicy** implementations to be notified that a *fat object* was inserted into the **Buffer** or that it was accessed from the **Buffer**, respectively.

4.3.2.4 Store

Management of *fat objects* in auxiliary memory is accomplished by the **Store** implementation. The **Store** behaves as a disk-based map for *fat objects*. Each **Store** implementation provides the **put** and **remove** interfaces. The **put** method requires the **Store** to persist the *fat object* and the **remove** requires the **Store** to support removal of the *fat object* from persistence by identifier. The persistence mechanism is up to the **Store** to implement, so long as the persistence adheres to the principles of *persistence durability* and *detachment*. These principles do not prohibit additional caching by the **Store**⁶, nor do they mandate non-volatile storage.

Definition 23. persistence durability: *Once a persistent fat object associated with an identity is stored, it can be subsequently retrieved by the same identity.*

Definition 24. persistence detachment: *Once a persistent fat object associated with an identity is removed, it can no longer be retrieved by the same identity.*

4.4 Implementation

This section details the implementation of the architecture specified in §4.3. JDiet is implemented for Java 2 Standard Edition (J2SE), version 5.0 [Sun Microsystems, Inc., 2006a], but may run in the J2SE Runtime Environment (JRE) version 5.0 or 6.0. Running within the common JRE follows the *portability* design principle.

⁶As the **Buffer** performs caching, any caching provided by the **Store** can be viewed as a 2nd layer cache

4.4.1 Logging

JDiet provides robust logging for debugging and error recovery. Logging is provided by both the RM and CTW components. JDiet utilizes the Simple Logging Façade for Java (SLF4J) [Quality Open Software, 2009] to perform logging, allowing consumers of JDiet to alter the logging provider as desired. SLF4J currently supports the following logging providers⁷:

- Log4j [Apache Software Foundation, 2007]
- Commons Logging [Apache Software Foundation, 2008b]
- JDK logging (`java.util.logging`)

4.4.2 Runtime Manager

The Runtime Manager (RM) is responsible for the runtime management of *fat objects*, particularly their buffering in main memory. The complete class design of RM is presented in Figure 16 in Appendix E.

4.4.2.1 Fat

The base interface `Fat` defines the characteristics and behavior required for an object to be managed by the RM. Each `Fat` object must implement the `Identifiable`, `Serializable` and `Sizeable` interfaces. The `Fat` interface is therefore no more than a type alias⁸ for the agglomeration of these three distinct interfaces.

⁷Refer to Quality Open Software [2009] for further details.

⁸Also known as a typedef in C and C++.

Identifiable

Each **Fat** object must support the mutation and access of its identifier. **Fat** objects being managed by the RM are identified by a unique 32-bit integer. Consequently, a maximum of 2^{32} fat objects can be managed⁹. These identifiers begin with the value of `Integer.MIN_VALUE` and continue until `Integer.MAX_VALUE`, at which point a runtime exception is thrown. As there currently exists no means to recycle identifiers for objects which are deallocated (see §4.4.2.2), systems likely to allocate more than 2^{32} objects within the same virtual machine instance are discouraged from using JDiet.

Serializable

All **Fat** objects must be capable of persisting their state to disk. Java Object Serialization (JOS) was chosen for its simplicity and native support for the JDiet-supported data types (see §2.3.4). As JDiet manages only primitive data, including **Arrays** and **Strings**, all **Fat** objects are inherently **Serializable** by default. JOS is not tied to a particular persistence technology, so a persistent **Store** must only be capable of persisting the serialized representation of **Fat** objects. Such a design maximizes flexibility of extension, but does incur performance penalties if the **Store** performs caching (see §4.4.2.5). Once a **Fat** object is serialized and placed into the **Store**, it must be deserialized when retrieved from the **Store** in the future. In cases in which the **Fat** object is resident in any cache(s) of **Store**, deserialization must still occur. However, such penalties are acceptable to adhere to the *extensibility* design principle.

⁹Such is the case even in 64-bit Java virtual machines [Venstermans et al., 2006].

Sizeable

The **Sizeable** interface provides that each **Fat** object is capable of describing its size. Similar in concept to the `sizeof` operator provided in some languages such as C, the **Sizeable** interface allows components of the RM, particularly sized-based eviction policies (see §4.4.2.4), to query the size of a **Fat** object. The **Sizeable** interface does not mandate the accuracy of the size returned, so **Fat** objects are capable of using heuristic measurements for their size (as discussed in §4.4.3.3).

4.4.2.2 Buffer Manager

The **BufferManager** is the class responsible for providing the interface to retrieve and store **Fat** objects. The implementation of the `load` and `manage` methods is primarily delegated to an underlying **Buffer** instance. The **BufferManager** provides synchronized, static access to a **Buffer** singleton. The primary consumer of these capabilities are thin objects whom are registering their associated **Fat** object or are acquiring access to a previously managed **Fat** object. Notably absent from the capabilities provided by the **BufferManager** is the ability to unmanage a **Fat** object. Such a capability is not provided in order to optimize performance, at the cost of disk space¹⁰.

During static initialization, the **BufferManager** creates a temporary space on disk which serves as the data directory for JDiet, particularly the **Store** for persistent data. The **BufferManager** uses the temporary directory specified for Java temp files, “java.io.tmpdir”, and creates a directory named “jdiet” below that. Modifying the value of the Java temporary directory environment variable

¹⁰Refer to §4.4.2.3 for more details.

allows the system designer to choose where JDiet stores its persistent data at runtime.

Similarly, during static initialization a shutdown hook on the runtime environment is established so that abnormal or abrupt terminations of the virtual machine can be detected (see Listing 1). This is accomplished by registering a listener on the current `Runtime` object using the `addShutdownHook` method. Upon termination of the virtual machine, the `BufferManager` cleans the JDiet temporary directory and closes the `Buffer`, allowing the `Buffer` and the underlying `Store` to release any file locks or other system resources.

Listing 1: BufferManager shutdown hook

```
1 // establish a shutdown hook that closes the buffer before
2 // shutting down
3 Runtime.getRuntime().addShutdownHook(new Thread() {
4     /**
5      * {@inheritDoc}
6      * @see java.lang.Thread#run()
7      */
8     public void run() {
9         try {
10             LOGGER.debug("VM shutting down, closing store");
11
12             try {
13                 // close the buffer on shutdown
14                 buffer.close();
15
16                 // delete working directory
17                 bufferDir.delete();
18             }
19             catch(CloseException be) { /* ignore */ }
20         }
21         catch(Exception e) {
22             LOGGER.error(e.getMessage(), e);
23         }
24     }
25 });
```

Code

The `BufferManager` provides two static methods for invocation by *thin objects*. The `manage` method is invoked during the construction of *thin objects* to register their associated `Fat` object with the `BufferManager`. The `BufferManager` assigns the `Fat` instance the next unique identifier. The next unique identifier is stored in the `nextFID` variable, which increments by one for each `manage` invocation. The identifiers begin at the value of `Integer.MIN_VALUE`. If the maximum number of valid identifiers has been reached (2^{32}), a `BufferException` is thrown (lines 14-17 in Listing 2). The `BufferManager` does not recycle identifiers for `Fat` objects that have been garbage collected (see §4.4.2.3). Once the `Fat` object has been assigned an identifier, it is inserted into the underlying `Buffer` instance (line 28). After successful insertion, the `BufferManager` returns the identifier of the `Fat` object. Refer to §4.4.3.3 for discussion of how this identifier is used by *thin objects* to establish a logical, rather than physical, reference to the `Fat` object. The *thin object* does not hold a strong reference to the `Fat` object after the `manage` method is invoked. Rather, access to previously registered `Fat` objects are made by invoking the `load` method.

The `load` method takes as input the identifier of a `Fat` instance, which must be an identifier assigned to a `Fat` instances by the `manage` method. This method, shown in Listing 3, simply delegates to the underlying `Buffer` instance to locate the `Fat` instance through the `find` method.

Both the `manage` and `load` methods are synchronized to prevent concurrent access to the `BufferManager`. Such synchronization avoids a race condition on the current identifier and state of the underlying `Buffer`. Should the `manage` method be invoked simultaneously by two threads, the value of `nextFID` could become inconsistent as it is shared statically among all threads. Furthermore, the

Listing 2: manage method implementation

```
1 public static synchronized int manage(Fat value)
2 throws BufferException {
3
4     if(nextFID == Integer.MAX_VALUE) {
5         throw new BufferException("Maximum amount of managed
6                                     Fat instances has been
7                                     reached");
8     }
9     else {
10         // assign the thin object a new id
11         int fid = nextFID++;
12
13         // assign ID
14         value.setID(fid);
15
16         LOGGER.debug("Managing fat object: {}", fid);
17
18         // insert the value in the buffer
19         buffer.insert(value);
20
21         // return ID
22         return fid;
23     }
24 }
```

Listing 3: load method implementation

```
1 public static synchronized Fat load(int fid)
2 throws BufferException {
3
4     LOGGER.debug("Loading fat object: {}", fid);
5
6     return buffer.find(fid);
7 }
```


Buffer is subject to similar race conditions. The only time at which the state of the **Buffer** can change is within the invocation of the **find** method, so synchronizing around its invocation ensures these race conditions are avoided. However, providing synchronization at this level of granularity limits the parallelism of the **BufferManager**. See §7.1 for a discussion of alternative implementations.

4.4.2.3 Buffer

The **Buffer** class is the central hub for the runtime management of **Fat** object. This class is responsible for managing a discrete set of **Fat** instances and interacting with an underlying **Store** based on the selections made by an **EvictionPolicy**. Each **Buffer** class manages a subtype of **Fat** instances, as bounded by the generic type parameter `<T>`. The number of **Fat** instances of type `<T>` is specified during the construction of each **Buffer**. As each **Buffer** must eventually evict instances to auxiliary memory, the **bufferDir** argument to the constructor identifies the directory in which the **Store** can place any persistence files. Each **Buffer** implements the **Closeable** interface such that the **BufferManager** can request the **Buffer** (and its underlying **Store**) to release any system resources, such as file pointers.

The **Store** instance the **Buffer** uses to perform eviction is identified using an environment variable. The **store_class** environment variable specifies the fully qualified name of the **Store** implementation (as described in §4.4.2.5)¹¹. Similarly, as a **Buffer** is constructed it identifies the **EvictionPolicy** to use when make the eviction decisions. The **Buffer** uses the **EvictionPolicy** specified by the **policy_class** environment variable. This environment variable denotes the

¹¹If a value is not provided, the **Store** implementation defaults to the **BDBStore** implementation.

fully qualified class name of an implementation of `EvictionPolicy` interface (as described in §4.4.2.4)¹².

Elements

`Fat` objects managed by the `Buffer` are either found in the `elements` collection (main memory) or in the underlying store `Store` (auxiliary memory), following the theorem of *monolocation* (Theorem 3). The choice of data structure for main memory storage is critical to performance and memory usage. The concerns of eviction and storage are separated such that the `elements` collection in the `Buffer` need not apply any particular ordering. Any ordering required to determine what `Fat` object to evict next is managed by the `EvictionPolicy`.

The `elements` collection is a unordered map of identifier to `Fat` instance. A map is critical to support lookup of `Fat` instances in $\Theta(n)$ time by identifier. As each access to a *fat attribute* within the *thin object* hits the `Buffer`, this operation must be as fast as possible. As the identifiers are integers, a map supporting integer keys is required. The `Map` implementations in the standard development kit could have been employed, however, these maps support only object types for keys and values. While Java supports autoboxing primitives, the operations to up- and down-cast to the appropriate object type are computationally expensive. While the `Integer` class can be used for the identifier, doing so needlessly wastes a minimum of 8 bytes of memory per identifier. This decision adheres to the *memory-driven* guiding principle. As the identifiers are immutable, wrapping them in an object is functionally unnecessary. As a result, the GNU Trove [GNU Trove, 2008] collections library is utilized. The collections provided by this library support primitive values natively with high performance.

¹²If a value is not provided, the `EvictionPolicy` implementation defaults to the LOF implementation.

Insert

The `insert` method, whose code is provided in Listing 4, inserts a `Fat` object of type `<T>` into the `Buffer`. This operation fails if the same `Fat` instance is being attempted to be inserted more than once. Lines 15-18 handle the case in which an insertion causes an element to be evicted. If the size of the `elements` collection reaches the maximum capacity of the `buffer`, a single element must be evicted by invoking the supporting method `evict`. Once room is made for the incoming `Fat` object, the instance is added to the `Buffer` by the `buffer` method.

Listing 4: insert method implementation

```
1 public void insert(T value) throws BufferException {
2
3     // has the value already been placed in this buffer
4     if(!elements.contains(value.getID())) {
5
6         LOGGER.debug("Inserting value: {}", value.getID());
7
8         // is the buffer full?
9         if(elements.size() == capacity) {
10             // if so, evict an object to make room
11             // for the new value
12             evict();
13         }
14
15         buffer(value);
16     }
17     else {
18         throw new BufferException("A value with the id " +
19                                 value.getID() + " already
20                                 exists in the buffer");
21     }
22 }
```

Evict

The `evict` method is responsible for evicting a single buffered element from the `Buffer` to the underlying `Store`. As shown in Listing 5, the `Buffer` requests that the `EvictionPolicy` select the next appropriate element to evict. Line 11 removes the selected element from the `elements` collection and line 17 places the evicted element into the `Store`.

Listing 5: evict method implementation

```
1 protected void evict() throws BufferException {  
2  
3     // select the item to evict  
4     T value = policy.selectEvictor();  
5  
6     // remove from the buffer  
7     elements.remove(value.getID());  
8  
9     LOGGER.debug("Evicting value: {}", value.getID());  
10  
11    try {  
12        // place the value in the store  
13        store.put(value);  
14    }  
15    catch(StoreException se) {  
16        throw new BufferException(se);  
17    }  
18 }
```

Buffer

The `buffer` method is invoked in either the `insert` or `find` methods when an element is being placed in the `Buffer` as either the result of a new addition or object fault, respectively. The `buffer` method, shown in Listing 6, is responsible for adding the `Fat` object to the `elements` collection. The `Fat` instance is associated with its identifier for fast retrieval. Line 11 performs an `insert` on the

underlying `EvictionPolicy` so the `policy` is made aware a new element so that it may place it in the appropriate eviction order (see §4.4.2.4 for more details).

Listing 6: buffer method implementation

```
1  /**
2   * Adds the given value to this {@link Buffer}
3   * @param value the value to buffer
4   */
5  protected void buffer(T value) {
6
7      // add to buffer
8      elements.put(value.getID(), value);
9
10     // add to policy
11     policy.insert(value);
12 }
```

Find

All access to **Fat** objects, resident in main memory or auxiliary memory is facilitated by the `find` method. Those **Fat** instance which have been evicted to the **Store** are swapped back into the **Buffer**. The situation in which this occurs is called an object fault. Shown in Listing 7, the `find` method handles the case of a cache hit from lines 13-18 and a cache miss (object fault) from lines 20-40. The `find` method first tests whether the **Fat** instance being queried exists in the `elements` collection by its identifier. As the `elements` collection is a map, this operation takes $\Theta(n)$ time. If a cache hit occurs, the `EvictionPolicy` is notified of the access to the **Fat** object with the `touch` method before the **Fat** instance is returned.

On the other hand, an object fault is handled in much the same was an insertion. Since the **Fat** object exists in the **Store** instead of the `elements` collection, it must be swapped into main memory. In order to do this, room

must be made in the **elements** collection if the main memory buffer is full. Lines 22-26 handle the case in which the **Buffer** is full and an existing **Fat** object must be evicted to make room for the element being swapped. Once again, the **EvictionPolicy** is responsible choosing which element is to be evicted. On line 30, the **Fat** instance being queried is retrieved from the underlying **Store**, which removes it from auxiliary memory persistence. Line 33 buffers the value by invoking the **buffer** method before returning the **Fat** object on line 35.

Garbage Collection & Orphans

The **Buffer** ignores the reclamation of **Fat** objects by the garbage collector. While the garbage collection of *thin objects* and **Fat** in the heap takes place, JDiet does not perform any special action on the garbage collection events of **Fat** objects. Detecting the reclamation of **Fat** objects can be accomplished in a variety of ways; however, each approach has drawbacks discouraging their use.

All Java classes are capable of detecting their finalization, the state in which the object is marked as finalized. Once an object has been finalized it may be reclaimed by the garbage collector [Zigman and Blackburn, 1999]. An object may be resurrected during the **finalize** method, preventing its reclamation [Boehm, 2003]. Consequently, an object can be made reachable during finalization. In such a case the **finalize** method is invoked only once [Gosling et al., 2005]. The **BufferManager** cannot rely on the finalization of an object to mean its lack of reachability. Overriding the **finalize** method also incurs performance penalties [Goetz, 2004; Goldstein et al., 2007], as a minimum of two garbage collection cycles are required for an object that overrides **finalize** to be reclaimed.

Java provides reference types which vary the behavior of how the garbage collector performs its reclamation (see §2.3.2). The **WeakReference** class pro-

Listing 7: find method implementation

```
1 public T find(int fid) throws BufferException {
2
3     // is the element in the buffer?
4     if(elements.contains(fid)) {
5         // touch the object (allow policy to update)
6         policy.touch(fid);
7
8         return elements.get(fid);
9     }
10    // its in the store
11    else {
12        // is the active space full?
13        if(elements.size() == capacity) {
14            // evict a thin object from the active space to
15            // make room for the thin object being faulted
16            evict();
17        }
18
19        try {
20            // fetch the value from the store
21            T value = store.remove(fid);
22
23            // put the value back in the buffer
24            buffer(value);
25
26            return value;
27        }
28        catch(StoreException se) {
29            throw new BufferException(se);
30        }
31    }
32 }
```

vides a reference whose existence is ignored in reachability calculations. If the `BufferManager` were to establish `WeakReferences` to `Fat` objects, their reclamation would not be prohibited. However, some notification of this reclamation would be required in order to ensure the synchronization of the `Buffer` contents with the heap. Coupled with the `ReferenceQueue` class, these types of references are commonly used to listen for the reclamation of an object [Leal and Ierusalimsky, 2004]. However, use of a `ReferenceQueue` imposes performance degradation and introduces synchronization issues. The `ReferenceQueue` is notified of a reclamation after the reclamation has occurred. Therefore, calls to the `get` method on an existing `WeakReference` may be made before the `Buffer` has handled the notification from the `ReferenceQueue`. Attempts to perform synchronization on the asynchronous reference reclamation events can limit throughput.

The performance degradation imposed by these two solutions is unnecessary. The `Buffer` maintains a strong reference to all `Fat` objects it is currently buffering in main memory. Any `Fat` object currently in the `Buffer` will remain in the `Buffer` until it is evicted, whether or not its associated thin object has been garbage collected. These types of `Fat` objects are referred to as orphans. Orphan fat objects may leak main memory temporarily, but leak auxiliary memory indefinitely. Orphan fat objects may also impact particular eviction policies, as discussed in §4.4.2.4.

Definition 25. Orphan fat object: *A fat object whose associated thin object has been reclaimed by the garbage collector.*

4.4.2.4 Eviction Policy

The `EvictionPolicy` interface describes classes capable of deciding which `Fat` objects to evict next from a `Buffer`. Each `EvictionPolicy` manages the eviction order of some subtype of the `Fat` interface. This interface exposes four methods that must be implemented by implementations of the interface.

The `construct` method is provided to enforce virtual implementation of a constructor, which is not provided for in Java. This construction establishes the maximum number of elements the `EvictionPolicy` must manage, allowing the `EvictionPolicy` implementations to optimize memory usage with fixed-size data structures. The `capacity` of the `EvictionPolicy` is the same as the `capacity` of the `Buffer` the `EvictionPolicy` manages the eviction order for.

To separate the concern of eviction ordering into the `EvictionPolicy`, the `insert` method must be invoked for each element taking part in the policy. There is a one-to-one correspondence between the number of elements in the `elements` collections and the elements managed by the `EvictionPolicy`. Elements are removed from the `EvictionPolicy` only through the invocation of the `selectEvictor` method, which is the core responsibility of the `EvictionPolicy`, as it selects the next appropriate element to evict from the `Buffer`. The order of eviction is left up to the implementations of the interface.

As some policies depend upon the frequency of use of elements, the `touch` method is provided so that implementations of the `EvictionPolicy` interface are notified when an element is accessed. The `touch` method is called for each `find` method invocation in the `Buffer`. Implementations may choose to ignore usage frequencies, but the commonality of this in many common policies, such as least-recently used and least-frequently used, makes such a notification appropriate for

the `EvictionPolicy` interface. Three types of `EvictionPolicy` implementations are implemented and available.

Usage-based

Usage-based policies are those that take into account the frequency or recency of use. Such policies leverage the `touch` method of the `EvictionPolicy` interface. A least-recently used (LRU) `EvictionPolicy` is implemented that inversely orders eviction based on the recency of access. This LRU implementation is implemented using the clock algorithm, which provides an approximation of LRU with high-performance and minimal bookkeeping (see §2.2.2). The `ClockLRU` class maintains references to `ClockEntry` objects, which keep track of the lock status of each element.

Queue-based

Queue-based policies are those that use a queue to order the eviction. These policies tend to be much simpler, and consequently less effective, than usage- and size-based policies. The `QueuePolicy` class provides a common abstraction over policies that use a `Queue` to order the elements. A queue-based policy using first-in first-out (FIFO) order is implemented by the `FIFO` class.

Size-based

A new policy is presented that takes into account the size of objects when performing eviction selection. While this policy shares inheritance with `FIFO` from `QueuePolicy`, the ordering of its queue uses the size of `Fat` objects. The `LOF` class implements the largest-object first ordering of elements using a `PriorityQueue` that orders its elements using a custom `Comparator`. This `Comparator` employs

the heuristic size of **Fat** objects returned by the `sizeOf` method of the **Sizeable** interface (as shown in Listing 8). If the first **Sizeable** object is larger than the second, the return value of the `compare` method is positive. If the first **Sizeable** object is smaller than the second, the return value is negative. If equal, the value of zero is returned.

Listing 8: SizeComparator used by L0F policy

```

1 private static class SizeComparator
2 implements Comparator<Sizeable> {
3
4     public int compare(Sizeable obj1, Sizeable obj2) {
5         // sort descending by size
6         return obj2.sizeOf() - obj1.sizeOf();
7     }
8 }

```

The L0F policy is predicated on the observation that when under memory constraints, JDiet is best served by evicting **Fat** objects with the largest footprint. However, L0F is subject to potentially undesired behavior when coupled with orphan **Fat** objects. Orphan **Fat** objects only become reclaimed when evicted to the **Store**. For queue- and usage-based eviction policies, it can be assured that any orphan **Fat** objects will be evicted relatively quickly. However, using L0F, orphan **Fat** objects of small size may remain in the **Buffer** as the L0F policy is sensitive to object size. Consequently, the L0F policy should be avoided in circumstances in which large **Fat** objects can prevent the reclamation of many orphan **Fat** objects of smaller size.

4.4.2.5 Store

The **Store** class provides an abstraction that describes the capability of implementing classes to persist **Fat** objects of type `<T>` in auxiliary memory. Each

Store manages persistent data on disk in the directory specified in the constructor. The disk resources required by each implementation of this abstraction are released by calls to the `close` method. Each **Store** behaves like a disk-based map, where each **Fat** object's persistent representation is stored and mapped against its identifier. The persistent state of a **Fat** object is the bytecode representation retrieved by serializing the instance using JOS.

BDBStore

The default **Store** implementation, **BDBStore**, uses the Berkeley Database [Oracle, 2009b] to persist **Fat** objects to disk. Berkeley Database (Java Edition) is an embedded high-performance non-relational database management system [Olson et al., 1999]. Storage is facilitated with the use of key-value pairs, so the Berkeley Database (BDB) is ideal for implementation of the **Store** abstraction¹³. BDB scales to millions of records, provides concurrency support, transaction management, and in-memory caching [Oracle, 2008].

In order for **Fat** objects to be persisted in BDB, the **BDBFat** subclass is provided (see Listing 9). The **BDBFat** class is a lightweight implementation of the **Fat** interfaces that bridges the **Fat** objects to the **BDBStore**. The **BDBFat** class uses annotations to expose it as an **Entity** with BDB¹⁴. Each BDB **Entity** is annotated with the `@Entity` annotation (line 12) and must provide a primary key with the `@PrimaryKey` annotation (line 16).

Upon construction, the **BDBStore** opens an **EntityStore** within the directory specified. The object through which queries to BDB are made is a **PrimaryIndex**, which is also created during construction (`index`). The `index` is appropriately

¹³Refer to Oracle [2009a] for a detailed comparison between BDB and traditional relational databases.

¹⁴Refer to Oracle [2008] for further details on BDB annotations

Listing 9: BDBFat class implementation

```
1 @Entity
2 public abstract class BDBFat implements Fat {
3
4     @PrimaryKey
5     private int id;
6
7     public BDBFat() { }
8
9     public int getID() {
10         return id;
11     }
12
13     public void setID(int id) {
14         this.id = id;
15     }
16 }
```

closed during the invocation of the `close` method. BDB supports in-memory caching to reduce disk I/O. Consequently, the `BDBStore` accepts an environment variable to configure the size of this cache. The `cache_size` environment variable specifies the size of the cache, measured as a relatively percentage of the heap size. The default value of `cache_size` is 10%. This cache is above and beyond the caching behavior provided by the `Buffer`, so the `BDBStore` as a layer two cache. Instead of caching the `Fat` objects, the `BDBStore` caches the serialized representation of each `Fat` object. Consequently, cache hits in this layer two cache are inherently more expensive, as the byte stream representation must be deserialized. A cache hit, however, remains must faster than a cache miss, as there is no disk I/O required.

The `BDBStore` implements the `put` and `remove` abstractions of the `Store`. The implementations of the `put` and `remove` methods are provided in Listing 10 and 11 respectively. BDB entities, by default, use JOS to serialize the rep-

resentation of objects for storage. Since `BDBFat` implements the `Fat` interface, and the `Serializable` interface, each `BDBFat` object can be stored natively in BDB without any additional marshalling or unmarshalling support. The `index` created to communicate with the database is utilized to `put` the `BDBFat` instance into the `BDBStore` (line 8 of Listing 10).

Unlike traditional uses of database management systems, each query of a `Fat` object removes (deletes) the record for the `Fat` instance, in accordance with the theorem of Colocation. As shown on line 11 of Listing 11, the instance retrieved from the `index` is subsequently removed before returning from the `remove` method. As was true for insertion, `BDBFat` objects are implicitly deserialized when the `get` method is invoked on line 9.

Listing 10: BDBStore put method implementation

```
1 @Override
2 public void put(BDBFat entity) throws StoreException {
3     try {
4         index.put(entity);
5     }
6     catch(DatabaseException de) {
7         throw new StoreException(de);
8     }
9 }
```

4.4.3 Compile-team Weaver

The Compile-time Weaver (CTW) in JDiet is implemented using the ASM bytecode engineering library [ObjectWeb, 2007]. ASM was chosen for its performance, simplicity of use, and well-documented application programming interface¹⁵. The complete class-level design of CTW is presented in Figure 16 in

¹⁵Refer to ObjectWeb [2007] for a complete API specification and developers guide.

Listing 11: BDBStore remove method implementation

```
1  @Override
2  public BDBFat remove(int id) throws StoreException {
3      try {
4          if(index.contains(id)) {
5              BDBFat val = index.get(id); // find it
6
7              index.delete(id); // remove it
8
9              return val; // return it
10         }
11         else {
12             throw new StoreException("A value with id '"
13                                     + id + "' does not
14                                     exist");
15         }
16     }
17     catch(DatabaseException de) {
18         throw new StoreException(de);
19     }
20 }
```

Appendix [E](#).

4.4.3.1 ASM Bridge

Several utility classes were implemented to supplement the ASM API. The abstract `ClassWeaver` class extends from the `ClassNode` class and allows for a composition pattern with other `ClassWeaver` instances. Each `ClassWeaver` maintains a `Collection` of `ClassNodes` it has generated, if any. Implementations of the `ClassWeaver` must provide the implementation for the protected method `mutate`.

Each `ClassWeaver` implementation can perform two types of bytecode manipulation: (1) weaving and (2) construction

Definition 26. Bytecode weaving: *Process of altering the instructions and structure of bytecode.*

Definition 27. Bytecode construction: *Process of constructing bytecode from some entity other than source code input.*

The `MethodWeaver` is a further abstraction of the `ClassWeaver`. The `MethodWeaver` implements the `mutate` method and provides another abstract method, `mutateMethod`. This class simply iterates over each method (`MethodNode`) in the class being manipulated and allows subclasses to handle mutation of each method individually.

4.4.3.2 Configuration

To support flexibility, the configuration of JDiet is managed externally with XML. An example configuration is presented in Listing [12](#)¹⁶. The `<Package>`

¹⁶The XML Schema for this configuration file can be found in Appendix [B](#).

and `<Class>` elements are used to hierarchically group `<Field>` elements. Each `<Field>` elements identifies to the JDiet CTW that a field should be extracted as fat. The case sensitivity of the field, class, and package names matches the case sensitivity of Java.

Listing 12: JDiet XML configuration structure

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Config>
3   <!-- Configures all classes within a single package -->
4   <Package name="edu.calpoly.app">
5     <!-- Configures a class and its fields -->
6     <Class name="A">
7       <!-- Identifies a field for thinning -->
8       <Field name="a"/>
9       <Field name="b"/>
10    </Class>
11    <Class name="B">
12      <Field name="c"/>
13    </Class>
14  </Package>
15 </Config>
```

4.4.3.3 Weavers

The CTW has three concrete weaver implementations, which are run in sequence. The CTW cannot complete its weaving and construct processes without a multistage parse, as all the entities exposed to the CTW must be known prior to weaving. The data flow of weavers is illustrated in Figure 6.

The JDiet bytecode engineering process begins with the configuration file, which configures the classes and fields subject to thinning. In order for the weaving and construction to correctly identify and extract the specified fields, the CTW must have all classes specified by the configuration exposed on the

Figure 6: Compile-time Weaver workflow

classpath. Furthermore, any classes the classes to be *thinned* derive from must be present in the classpath¹⁷.

Class Mapper

The `ClassMapper` does not perform any mutation of the `ClassNodes` it iterates over, but rather extracts identification information about each `ClassNode` for future use by the downstream weavers and constructors. This information includes the name of the class, its fields, and a mapping of what classes and fields are subject to mutation, as guided by a configuration. As this information is read upfront, downstream weavers and constructors can query the inheritance hierarchy of a class being *thinned*.

Fat Extractor

The `FatExtractor` is a joint weaver/constructor that processes input classes and performs the fat extraction, as guided by the configuration. Each class in the ASM tree is traversed and cross-referenced with the `<Class>` elements in the configuration. For each class configured for thinning, all attributes specified by the `<Field>` elements are extracted. If an attributed specified by a `<Field>`

¹⁷This is due to constructor weaving, which requires knowledge of the structure of the parent class.

element does not exist, it is safely ignored. The fat extraction process for each class proceeds in three steps:

Step 1: Fat class construction

During the *fat class* construction process, a new class is constructed to serve as a container for all the *fat attributes* to be extracted. There is a one-to-one relationship between a *fat class* and its associated *thin class*, even with respect to inheritance. The access modifier of the class is set to public (`ACC_PUBLIC`) and the super flag is set for backwards compatibility (`ACC_SUPER`)¹⁸. The `ACC_FINAL` flag is also set as the *fat class* is not designed for extension. To ensure no name collisions, the *fat class* is given the same name as the class being *thinned*, but appended with the suffix “`$_FAT`”¹⁹. For example, a class named `Foo` would have a *fat class* created with the name `Foo$_FAT`.

As JDiet is currently based on the Berkeley Database `Store` all *fat classes* derive from `BDBFat`. The *fat class* extends the `BDBFat` abstraction to inherit the implementation of the `Fat` and `Identifiable` interfaces without byte-code modification. A default constructor is added with package visibility (`ACC_PACKAGE`) to allow construction from any class in package scope. This prevents construction by external classes not managed by JDiet. The byte-code instructions of all *fat class* constructors is provided in Listing 13. Line 2 loads the `this` reference on the top of the stack for the super constructor call on line 3. Like all constructors, the fat class constructor returns void.

As there exists to no built-in operator in Java to query the dynamic size of

¹⁸See [Sun Microsystems, Inc. \[2006b\]](#) for an explanation of the use of the `ACC_SUPER` access flag.

¹⁹The use of the currency symbol (\$) is common for classes created implicitly by the compiler, such as anonymous classes.

Listing 13: Fat class constructor

```
1 <<init>()V>
2 aload_0
3 invokespecial <edu/calpoly/jdiet/fat/impl/BDBFat.<init>>
4 return
```

an object, the `sizeOf` method is implemented to support size-based eviction policies. To maximize performance, the size of a *fat class* is pre-computed at compile time using heuristic weights. The weights used to calculate the size of a *fat class* are presented in Table 2. The weights roughly mirror the size of the data types in the Java language²⁰, with three notable exceptions. The boolean data type takes only a single bit to represent its value, but most Java virtual machines are octet aligned, so consuming a single bit is as costly as consuming an entire octet. **String** and **Array** data types, although considered primitive data types, can grow or shrink dynamically. Therefore, their size cannot be calculated precisely at compile-time. To counteract their dynamic size, these types are given considerably more weight than the other primitive data types. The size of **Arrays** and **Strings** can be calculated more precisely at runtime, but there is an unexpected consequence associated with this calculation—as the data values of these types can change dynamically, each **EvictionPolicy** that uses the **Sizeable** interface must resort the eviction priority list on each data change. This resorting can be computationally expensive when the buffer size is large²¹. A notification mechanism to alert the **EvictionPolicy** of changes in these data types is currently not supported and poses potentially detrimental performance implications. Consequently, the heuristic weights

²⁰The number of octets required to hold a data value on a 32-bit virtual machine.

²¹Even the fastest sorting algorithms are $\Theta(n \cdot \log(n))$.

provide reasonable estimates for consumers of the `Sizeable` interface with optimal performance.

Table 2: Heuristic Data Type Weights

Data Type	Weight
boolean	1
byte	1
char	2
short	2
int	4
double	8
long	8
Array	60
String	60

Step 2: Fat attribute extraction

Once the signature of the class has been constructed, the `FatExtractor` iterates over all the fields subject to thinning. These fields are removed from the source class and placed into the fat class. The name and descriptor of each field remains the same.

Step 3: Logical reference from thin class to fat class

`Fat` objects are inherently constructed when instances of their associated *thin classes* are constructed. For each constructor in the *thin class*, an instantiation of the *fat class* extracted is made. It is at this point that the `manage` method of the `BufferManager` is invoked to register the `Fat` object and receive its identifier. This identifier is stored in the `Fat` object as well as the *thin object*. With the *fat attributes* removed from the *thin class*, a logical reference must be established between the two classes. This reference is established in the constructor of the *thin class*.

Each constructor of the *thin class* is woven according to the template in Listing 14. Only base constructors are woven, that is, constructors that do not call constructors in the same class (i.e. the **this** constructor call). As the super constructor call must be the first method invocation in any constructor, the weaving takes place after the special method invocation of the `<init>` method. Beginning on line 4, the `this` reference is pushed on the stack. Between lines 5 and 7, a new instance of a fat object is constructed using the default constructor. This fat object reference is passed to the static `manage` method of the `BufferManager`. The `manage` method is responsible for storing the `Fat` object appropriately and assigning a fat identifier. This fat identifier is stored locally in the thin object on line 9.

Listing 14: Thin class constructor weaving

```

1 <<init> (... )V>
2 aload_0
3 invokespecial <[SuperClass].<init>>
4 aload_0
5 new <[Class]$_FAT>
6 dup
7 invokespecial <[Class]$_FAT.<init>>
8 invokestatic <edu/calpoly/jdiet/buffer/BufferManager.manage>
9 putfield <[Class]_.--fid>
10 ...
11 return

```

This logical pointer is dereferenced each time access to any *fat attribute* is made. A utility method with private visibility, `loadFat`, is inserted into the *thin class* to facilitate this dereferencing process. The general structure of the instructions is shown in Listing 15. The symbol `<Class>` represents the name of the class being thinned. The first two lines retrieve the current value of the fat reference identifier. This identifier is passed to that static

`load` method of the `BufferManager`. As the return type of the `load` method is an instance of the `Fat` interface, as cast is performed to ensure type safety on line 5.

Listing 15: loadFat method construction

```
1 <__loadFat ()L[Class]$FAT>
2 aload_0
3 getfield <[Class].__fid>
4 invokestatic <edu/calpoly/jdiet/buffer/BufferManager.load>
5 checkcast <[Class]$FAT>
6 areturn
```

For each *fat attribute* extracted, a pair of accessor and mutator methods are inserted into the thin class. These methods are constructed using the bean style naming convention and are prefixed with “__” to avoid name collisions with common accessor and mutator method name²². For example, a field named `position` would have methods `setPosition` and `getPosition` created respectively.

Instead of forcing the *thin classes* to adhere to the Java bean style, the `CodeWeaver` provides this implementation automatically. Each accessor and mutator method invokes the `loadFat` method woven into the thin class by the `FatExtractor` before `.` These additional methods are constructed with public visibility, allowing them to be invoked publicly. The template of instructions for each method pair created is outlined in Listings 16 and 17. In both cases, the methods are no more than proxies for the field retrieval or assignment.

An important feature of the implementations of these accessor and mutator methods is that strong references to the fat objects are not retained. The

²²The name of attribute in the name of the accessor and mutator is changed to a leading uppercase, to adhere to the common lower camel-case naming convention.

Listing 16: Fat accessor method

```
1 <__get [ Attribute ] ( ) [ DataType ] >
2 aload_0
3 invokespecial <[ Class ]. __loadFat >
4 getfield <[ Class ] $ _FAT . [ attribute ] >
5 areturn
```

Listing 17: Fat mutator method

```
1 <__set [ Attribute ] ( [ DataType ] ) >
2 aload_0
3 invokespecial <[ Class ]. __loadFat >
4 aload_1
5 putfield <[ Class ] $ _FAT . [ attribute ] >
6 return
```

reference to the `Fat` object, acquired in line 3 for both listings does not escape the stack frame. This ensures that every logical access of a fat object attribute must invoke `loadFat`.

Code Weaver

The bytecode mutation process is not complete until access to the extracted *fat attributes* is addressed. As the *fat attributes* could have previously been accessed publicly by external classes or privately by the *thin class*, all access code paths must be modified. This is the responsibility of the `CodeWeaver`, which is a specialization of a `MethodWeaver`.

All non-static field access instructions (`getfield` and `setfield`) within any method body (including constructors, initializers, static methods and instance methods) to the *fat attributes* are modified so that they invoke the corresponding *fat attribute* accessor or mutator. These changes affect not only the thin class itself, but any other class that previously accessed fields of a class publicly (with-

out encapsulation). Consequently, in order for the CTW to correctly thin classes, all classes accessing *fat attributes* must be exposed to the weaver.

4.4.3.4 Ant Integration

To ease the bytecode manipulation process, the `Weave` class allows the JDiet CTW to be run with the Ant build tool [Apache Software Foundation, 2008a]. The `Weave` class extends the `Task` base class to provide a custom Ant tag for use in any Ant build script. The usage of the `Weave` task is outlined in Listing 18.

Listing 18: Weave task usage

```
1 <taskdef name="weave" ...
2         classname="edu.calpoly.jdiet.weaver.ant.Weave">
3 ...
4 <weave destDir="..." config="..." >
5     <fileset dir="..." includes="**/*.class"/>
6 </weave>
```

To import the `Weave` task, a `taskdef` or `typedef` must be specified with the fully qualified classname of the `Weave` class²³. The task requires two attributes and a nested element. The first attribute specifies the destination directory in which to write the output class files. The `destDir` specifies the root directory, so classes with packages will be written in subpackages beneath this root directory. The `config` attribute specifies the path and name of the file specifying the configuration of JDiet. The file specified must be a valid JDiet XML configuration file. The `Weave` task accepts a single nested `fileset`, which contains all the class files to expose to the weaver. The `Weave` task performs the following operations:

1. **Reads in a configuration file.** The configuration file is unmarshalled

²³The JDiet JAR file must therefore be on the current Ant classpath or specified by a classpath or classpathref value. See Apache Software Foundation [2008a] for further information.

into an object model using the XStream [[Codehaus, 2008](#)], a lightweight XML serialization library.

2. **Exposes the class files to the CTW.** A `Weaver` is constructed with the parsed configuration.
3. **Locates class files to expose to the CTW.** The `fileset` specified is evaluated and each file matching the include pattern is passed to the `Weaver`.
4. **Runs the CTW.** The `Weaver` is executed and the weaving/construction process proceeds.
5. **Writes any generate fat classes to disk.** All generated classes constructed by the CTW are written to disk under the `destDir` root directory. Any existing files with the same name as the output classes are overwritten without warning.

5 Evaluation

A quantitative evaluation of the contribution of this paper is conducted. Kitchenham et al. [1994] define a formal experiment as “a means of testing, using the principles and procedures of experimental design, whether a hypothesis can be confirmed.” JDiet is evaluated with respect to its primary goal: running a memory-constrained application within a memory footprint much less than that originally required. A representative application requiring significant memory to operate is selected and a statistical analysis of the following hypotheses is conducted.

Hypothesis 1: Success

- *Null*: The success of an application is not increased by JDiet.
- *Alternate*: The success of an application is increased by JDiet.

Definition 28. success: *The complete execution without abnormal terminal.*

Hypothesis 2: Memory consumption

- *Null*: The memory consumption of an application is not reduced by JDiet.
- *Alternate*: The memory consumption of an application is reduced by JDiet.

Definition 29. memory consumption: *The maximum memory consumed by the runtime environment, including all runtime data areas.*

5.1 Benchmark

Software benchmarks are a well-accepted technique to perform software evaluation and testing [Anderson, 1990; Blackburn et al., 2006; Carey et al., 1993]. The Xerces Document Object Model (DOM) Parser [Apache Software Foundation, 2005] is selected as a suitable benchmarking application. DOM is a language independent model for XML data [W3C, 1998], with multiple implementations in various languages. DOM objectifies XML in a tree structure, with nodes representing elements, attributes and text alike. Unlike event based parsers, such as the Simple API for XML (SAX), or stream parsers, such as the Streaming API for XML (StAX), DOM provides intuitive iteration and navigation through these nodes.

DOM is well-known for its demanding memory qualities, but it still used widely used for its simplicity and XPath [W3C, 2007] support. Unlike event- and stream-based parsers, the construction of a DOM document is expensive, with regards to both time and memory. Most notably, the size of a DOM tree can be as much as ten times the size of the document being parsed [Wang et al., 2007], despite features such as deferred loading. As such, systems employing a DOM parser are likely candidates for memory-constrained systems.

The benchmarks used to assess the applicability of the proposed tool includes two types of DOM manipulations: (1) parsing and (2) traversal. In order to facilitate these benchmarks, XML documents of varying sizes are generated using the *xmlgen* data generator component of the XMark benchmark tool [Schmidt et al., 2002]. Other XML benchmarks considered included ToXgene [Barbosa et al., 2002], X007 [Li et al., 2001], and XBench [Yao et al., 2003]. XMark is chosen for its simplicity of use, extensive documentation and ease of generation. Most XML

benchmarks are oriented towards XML querying, but *xmlgen* is a comprehensive XML data generator that can produce documents that are a replicative factor of a base document. The XML documents produced by *xmlgen* are indicative of common-place XML documents and include a mix of elements, attributes and textual content that mimic an auction website. Documents can be generated ranging anywhere from tens of kilobytes up to gigabytes. The standard, or one factor, document is roughly 145 MB. Barbosa et al. [2002] provide a detailed description of how these documents are generated and the XML constructs they contain.

Keeping in mind the demanding memory requirements of a DOM parser, parsing the base document created by *xmlgen* requires roughly 600 MB of main memory to process. While most modern computer have this much memory available, the interaction of other processes within an operating system (including other memory requirements of the application parsing the document) often require the use of virtual memory swap space. Such a situation often leads to frequent page misses and thrashing, hindering performance, and eventually in many cases crashing the application. In the case of Java, this occurs when a `java.lang.OutOfMemoryException` is thrown¹. Consequently, an application utilizing DOM parsing is well-suited for the application of JDiet.

5.2 Experimental Design

The experiment to test the hypotheses is performed on a control and experimental group. The control group includes the unaltered DOM parser implemen-

¹This exception can be thrown in cases in which virtual memory has yet to be entirely consumed; in fact, a `java.lang.OutOfMemoryException` may be thrown in cases in which the operating system fails to expand the heap space within a certain allowable limit, e.g. during garbage collection

tation provided by Xerces. The experimental group consists of the same parser implementation subject to JDiet. As a result of performance analysis using hprof [O’Hair, 2004] and JProfiler [ej-technologies, 2009], a configuration identifying the textual content of text nodes is used. The complete configuration used in the experimental group is provided in Listing 19. This configuration selects the `data` attribute of the `CharacterDataImpl` class as the single *fat attribute*. JDiet currently supports only fixed buffer sizes, as specified by the number of total objects. Consequently, to maintain consistency between test cases, a relative percentage of the number of objects being *thinned* in the source document is used. The experimental group tests are run using a buffer size that is 5% the total number of text nodes in each input file. The cache size used for the `BDBStore` is 5% the heap size. For both the control and experimental groups, each test is replicated three times to reduce variability of the testing environment.

Listing 19: Experimental Group JDiet Configuration

```
<?xml version="1.0"?>
<Config>
  <Package name="edu.calpoly.jdiet.test.xml.app
              .xerces.org.apache.xerces.dom">
    <Class name="CharacterDataImpl">
      <Field name="data"/>
    </Class>
  </Package>
</Config>
```

5.2.1 Independent Variables

Time. The aggregate time required to complete a benchmark test, measured in seconds (s). Should a benchmark fail to complete, as defined by definition 28, the time is undefined.

Memory. The aggregate memory required for a benchmark test, measured in megabytes (MB). Should a benchmark fail to complete, as defined by definition 28, the memory usage is undefined.

5.2.2 Dependent Variables

Input size. The size of the benchmark XML file given to the DOM benchmarks.

The files generated using the *xmlgen* data generator range in size from 34.91 KB to 290.74 MB (see Schmidt et al. [2002] for further details). As the size of the input files increase, their relative makeup remains nearly constant, with approximately 35.5% element nodes and 64.5% text nodes. See Appendix F for detailed characteristics.

Heap size. The maximum amount of memory that can be allocated to the heap for the Java virtual machine execution² (measured to the nearest MB). The maximum heap sizes include 32 MB, 64 MB, 128 MB, 256 MB, 512 MB, and 1024 MB.

Workload. The type of workload, or process, the benchmark is performing. The following workloads are performed by the benchmarks:

- **Parsing** performs a parse of the input file into a DOM document tree.
- **Iteration** performs parsing and runs a depth-first iteration over all nodes, inspecting neither the elements content nor attributes.

Eviction Policy. The eviction policy used by the **Buffer** when determining which entries to evict after an object fault. The following eviction policies are available (refer to §4.4.2.4 for the implementation details of each policy):

²Specified during the construction of the Java virtual machine using the -Xmx flag. See Sun Microsystems, Inc. [2006b] for further details.

- FIFO: first-in first-out
- LRU: least-recently used
- LOF: largest object first

5.2.3 Hardware Configuration

The evaluation is performed on a machine with the following configuration:

- Mac OS X 10.5.6
- Java 2 Standard Edition (J2SE) JRE 1.5.0_16
- 2.4 GHz Intel Core 2 Duo (4MB L2 Cache)
- 2 GB 667 MHz DDR2 SDRAM
- 149.05 GB S-ATA HD

5.3 Results

The results of the benchmarks of all input files successfully completed within a virtual machine with maximum heap sizes of 32 MB, 256 MB, and 1024 MB are detailed in the following sections. A discussion and analysis of these results proceeds in §5.3.5.

5.3.1 32 MB Maximum Heap Size

The parse memory usage results for the 32 MB maximum heap size are presented in Figure 7. The memory consumption of the control group is roughly five times the size of the input document. As a result, allocating a maximum heap size

allows the successful completion of only the first three input files, the maximum of which is 5.5 MB. On the other hand, each eviction policy used by the experimental group results in five of the input files to be successfully parsed. For the 5.5 MB input file, JDiet consumes 26.27%, 38.16%, and 12.10% less memory for the FIFO, LOF and LRU policies respectively. The percentage memory gained through each input file size increase appears to be non-linear, as the decreased memory usage in the 2.8 MB input file is 33.91%, 24.15%, and 11.58% for the FIFO, LOF, and LRU respectively. All three policies for the experimental group are capable of successfully parsing an input document 264.36% larger than the control group.

Figure 7: Parse Memory Usage with 32 MB Max. Heap Size

The parse time results for the 32 MB maximum heap size are presented in Figure 8. Parsing the 5.5 MB input file takes approximately 700% longer for

the experimental group using JDiet than without. Only until the largest input file capable of being parsed using a 32 MB heap size does there exist significant differences between the execution times of each experimental group policy. The LOF policy is the fastest, taking 9.92 seconds to parse the 14.54 MB input file, which is 7.2% faster than FIFO and 13.26% faster than LRU.

Figure 8: Parse Total Time with 32 MB Max. Heap Size

The iteration time results for the 32 MB maximum heap size are presented in Figure 9. For the largest input file the control group is capable of parsing and iterating, it takes approximately 6% longer for the control group to perform iteration as compared to each policy of the experimental group. With the notable exception of the 14.54 MB input file, the iteration time was roughly constant for each eviction policy. The experimental group with the LOF policy appears to have outlier data, as the first four input files have generally equivalent results

across each policy. Such an outlier could be the result of hardware variability during testing and is discussed in §5.3.5.

Figure 9: Iteration Total Time with 32 MB Max. Heap Size

The complete listing of results for the 32 MB maximum heap size can be found in Tables 4, 5, and 6 located in Appendix F.1.

5.3.2 256 MB Maximum Heap Size

The parse memory usage results for the 256 MB maximum heap size are presented in Figure 10. The smallest three input files demonstrate a smaller memory footprint for the control group, with the second input file (2.8 MB) providing the maximum percentage difference between the control and experimental groups at roughly 86.88%. However, by fifth input file, there are marginal differences

between the memory consumption of both groups. The input file size of 14.54 MB appears to be the inflection point for the memory usage gains made by the experimental group overcoming the control group. As the input size increases, the memory consumption of the experimental group show a decreasing trend in percentage compared to the control group. The non-linear trend observed in the memory usage savings by the experimental group over the control group in the 32 MB heap size test cases (see §5.3.1) are confirmed. For the LOF eviction policy, the control group consumes 28.86%, 33.40%, and 64.88% more memory to parse the 29.29 MB, 43.67 MB, and 58.13 MB documents respectively. The percentage difference in savings between each input file is 15.73% and 94.25%, which suggests a non-linear trend. The largest input the control group is successfully capable of parsing is the 58.13 MB document. Conversely, all but the LRU policy experimental group are capable of parsing up to the 131.01 MB input, an increase of 225.37%.

The parse time results for the 256 MB maximum heap size are presented in Figure 11. In similar trends observed in the 32 MB heap size data set (see §5.3.1), the experimental group takes about 820% longer to parse the 58.13 MB input file, the maximum input file the control group is capable of parsing. The relative performance of each policy within the control group appears to grow in disparity with the size of the input document. For the 116.44 MB input file, the LRU and FIFO policies take 17.03% and 10.74% longer than LOF, respectively.

The iteration time results for the 256 MB maximum heap size are presented in Figure 12. The total time required to perform the depth-first iteration on the input documents is relatively equal until the seventh input document (43.67 MB). The difference in iteration time is a mere 5.36% worse for the 43.67 MB input file, but a noticeable 34.38% worse for the 58.13 MB input document (an

Figure 10: Parse Memory Usage with 256 MB Max. Heap Size

Figure 11: Parse Total Time with 256 MB Max. Heap Size

absolute difference of 0.258 seconds). Despite the inability for the LRU policy to complete the 131.01 MB test case, the performance between each policy of the experimental group appears to remain relatively similar. Refer to §5.3.5 for a discussion of possible causes of this anomaly.

Figure 12: Iteration Total Time with 256 MB Max. Heap Size

The complete listing of results for the 256 MB maximum heap size can be found in Tables 7, 8, and 9 located in Appendix F.2.

5.3.3 1024 MB Maximum Heap Size

The parse memory usage results for the 1024 MB maximum heap size are presented in Figure 13. As noted in both the results from the 32 MB and 256 MB benchmarks (§5.3.1 and §5.3.2), the results from the 1024 MB maximum

heap size show an inflection point near the smaller input files for which the control group has lower memory usage than any of the policies of the experimental group. For this case, the inflection point appears at the 58.13 MB input file. For the remaining larger input file sizes, the experimental group outperformed the control group, ranging from 20.1% memory improvement in the 87.57 MB input document to 38.56% in the 144.96 MB input document. Again, there exists a positive trend between the relative memory usage decrease as the input file size increases.

Figure 13: Parse Memory Usage with 1024 MB Max. Heap Size

The parse time results for the 1024 MB maximum heap size are presented in Figure 14. For those input files both the control and experimental groups successfully parsed (all but 290.74 MB), the control group performed significantly faster. For the 144.96 MB input file, the control group outperformed the experimental

groups by an approximate average of 760%. This trend confirms the results observed in the 32 MB (§5.3.1) and 256 MB (§5.3.2) data sets. However, unlike previous trends an anomaly is present for the parse time when using the LRU policy, between the 43.67 MB and 102.26 MB input files. During this period, the LRU policy displays significantly longer parse times than the other replacement policies. This anomaly seems to correct itself for the 116.44 MB input files and above.

Figure 14: Parse Total Time with 1024 MB Max. Heap Size

Iteration time results for the 1024 MB maximum heap size are presented in Figure 15. Iteration times remain roughly equivalent between the control and experimental groups until the 58.13 MB input file. For all successful benchmarks beyond this, the experimental group performs noticeably faster, ranging from 6% (72.21 MB) to 34.18% (58.13 MB).

Figure 15: Iteration Total Time with 1024 MB Max. Heap Size

The complete listing of results for the 1024 MB maximum heap size can be found in Tables 10, 11, and 12 located in Appendix F.3.

5.3.4 Analysis

5.3.4.1 Success

The null hypothesis of success is rejected in favor of the alternate hypothesis. There exists substantial evidence which strongly suggests that the success of an DOM-based application can be increased with the use of JDiet. For the 32 MB maximum heap size the control group (without JDiet) is able to parse only the three smallest files, for a success rate of 20%. On the other hand, the experimental group, spanning all three eviction policies, is able to parse the smallest five input files, for a success rate of 33.33%. For the 256 MB maximum heap size, the control group is able to parse eight of the fifteen input files, for a success rate of 53.33%. Whereas, two of the three policies used in the experimental group can parse five more input files, for an improved success rate of 86.67%. The evaluation results conclusively indicate that the success of parsing the input benchmark files is increased with the application of JDiet.

5.3.4.2 Memory Consumption

The null hypothesis of memory consumption is rejected in favor of the alternate hypothesis. There exists strong evidence suggesting JDiet reduces the memory usage of a DOM-based application. Those not favored in all cases, there exist several benchmarks in which the memory reduction are significant. For the 32 MB heap size, a maximum memory reduction of 38.2% was observed in the 5.5 MB input file with the use of the LOF policy. For the 256 MB heap size, a

maximum reduction of 64.88% is again achieved by the LOF policy. In general, there appears to be an inflection point for which the use of JDiet begins having a strong affect on memory consumption. For all test cases, this inflection point appears toward the lower end of the input file sizes, suggesting JDiet provides benefits more often than not.

The reduced memory footprint allowed the experimental group to parse input documents of much larger size than it could without JDiet. With the most constrained heap size of 32 MB, an input file 264% larger was successfully parsed by all policies in the experimental group as compared to the control group. Similar results were observed in the 256 MB data set, allowing a 225% larger document to be parsed. The observed results suggest that memory usage across each eviction policy for the experimental group varies³. In general, the LRU policy requires more memory than the FIFO and LOF policies. This is expected, as the LRU policy entails more bookkeeping to record the last use time. However, the difference between FIFO and LOF is not as expected; as a single `String` attribute is *thinned*, the ordering of eviction for LOF should be equivalent to FIFO⁴. The variation in results could be the result of the difference in internal management and performance of the `Queue` collection implementations. The FIFO policy employs the `LinkedList` implementation for the `Queue` interface while LOF uses the `PriorityQueue`.

³As the sizes of the input document were discrete, these savings are approximations.

⁴Due to heuristic weights, each `Fat` object will have the same value returned from the `sizeOf` method.

5.3.5 Discussion

5.3.5.1 Performance

While both hypotheses have been confirmed using benchmarks performed on the control and experimental group, the cost for such savings is clearly visible. Performance suffers significantly during the parsing process, in order to maintain a reduced footprint, JDiet evicts *fat objects* to disk, incurring the penalty of disk I/O. For the worst case in each data set, the average performance cost for the parsing process is approximately 760%. However, while in most cases the upfront parsing cost was significant, there were unexpected benefits during the iteration process. Illustrated well in the iteration times of the 1024 MB maximum heap size (see Figure 15), the performance benefit is as much as 34%. Such a performance improvement is a likely result of less frequent garbage collection as the heap size grows.

5.3.5.2 Variability & Outliers

There exists some degree of variability in the measurements taken during the benchmark testing. The memory consumption of the virtual machine is monitored as closely around each test case as possible with the use of the `Runtime` class⁵. Despite the accuracy of these metrics, the times at which the memory usage snapshots are taken depends on the random scheduling of garbage collection process (and the context switching between multiple threads within the virtual machine⁶). Replication is used to counteract such variability and imprecision. A potential outlier in the 32 MB iteration time results (see §5.3.1) is identified

⁵Specifically, the memory consumption is calculated using the difference between the values of `Runtime.totalMemory()` and `Runtime.freeMemory()`.

⁶The garbage collection process runs in a separate thread.

and may be caused by such scheduling variability. Such outliers are likely to be exacerbated in shorter run times and smaller test cases.

The observed parse times results for the 1024 MB maximum heap size demonstrate an anomaly for the LRU policy between the 43.67 MB and 102.26 MB input files. This anomaly may be explained by concurrent load undertaken during these tests by the benchmark machine. Operating system background tasks, such as indexing, backups, and other various processes may have contributed to the short period of skewed results. Such multitasking would explain the more consistent results observed for the 116.44 MB input files and above.

6 Related Work

Persistence is a commonly used means to offload data from main memory to auxiliary memory. Persistence is defined as the “period of time for which the object exists and is usable” [Atkinson and Morrison, 1995]. It has been approximated that typically 30% of all code in modern programming languages is concerned with transferring data to and from secondary storage [Atkinson et al., 1983; Grimstad et al., 1999]. These applications have been termed persistent application systems (PAS) [Atkinson and Morrison, 1995]. The most common forms of persistence include the explicit use of a database, object-relational mapping, object-oriented databases, or file-persistence [Bläser, 2006; Ortín et al., 2004]. Applications prepare for persistence using one of three levels of persistence definition [Al-Mansari et al., 2007]:

- **Type-level** preparation includes adding information to types (classes) or naming conventions
- **Code-level** preparation includes code that identifies what types should be made persistent
- **Object-level** preparation includes dynamic approaches identifying persistent entities at run-time

6.1 Memory Leak Tolerance

A memory leak manifests in an unmanaged language when memory is allocated and not explicitly freed. Other types of memory problems include freeing memory twice, dangling pointers, and lost pointers [Bond and McKinley, 2006]. A dangling pointer occurs when memory is dereferenced after it is freed, while a lost

pointer is a condition in which memory previously allocated loses all references to it and can no longer be freed [Jump and McKinley, 2007].

Even though managed languages can prevent many of these types of memory leaks, they do not prevent them all. A common misconception is that managed languages provide “worry-free” memory management [Dingle, 2004]. The automated memory manager does not relieve the developer from all memory issues [Goldstein et al., 2007]. In a managed language, memory is leaked when it is allocated but no longer used, but prevents the memory management system from reclaiming it [Chilimbi and Larus, 1998; Jump and McKinley, 2007]. While managed languages prevent lost pointers, they do not prevent unused references. These types of leaks are often referred to as loitering objects [Goldstein et al., 2007], unnecessary references [Jump and McKinley, 2007] or zombie reference [Rayside and Mendel, 2007]. These references are not required for correct functioning of the program but have been neglected to be released [Bond and McKinley, 2006; Rayside and Mendel, 2007]. Because typical memory managers approximate liveness of memory by reachability [Bond and McKinley, 2008], they do not reclaim unused, but reachable memory [Goldstein et al., 2007]. Shaham et al. [2001] refer to the time it takes unused memory to become unreachable as “drag.”

While some memory leaks can manifest themselves quickly, some can occur in specific environments and take weeks or even months to cause problems [Bond and McKinley, 2006]. Unlike other types of memory errors, memory leaks are the hardest to detect and may not result in any immediate or diagnosable symptoms [Hasting and Joyce, 1992]. In managed languages, the increased memory space consumed by leaked memory significantly hinders performance [Bond and McKinley, 2006, 2008; Rayside and Mendel, 2007], but may not crash the pro-

gram immediately. While memory leaks in managed languages are much less frequent, they are often more severe [Dingle, 2004]. Common causes of memory leaks in Java include not unregistering listeners [Dingle, 2004], string inefficiencies [Kawachiya et al., 2008] and structural overhead [Mitchell and Sevitsky, 2007].

Common leak detection tools and techniques include Sleigh [Bond and McKinley, 2006], SWAT [Hauswirth and Chilimbi, 2004], Cork [Jump and McKinley, 2007], LeakBot [Mitchell and Sevitsky, 2003], Purify [Hasting and Joyce, 1992], Valgrind [Nethercote and Seward, 2007], Safe-Mem [Qin et al., 2005], FindLeaks [Chen and Chen, 2007], cyclic memory allocation [Nguyen and Rinard, 2007], object ownership profiling [Rayside and Mendel, 2007], and container profiling [Xu and Rountev, 2008]. While these tools and techniques focus on detection and possible prevention of memory leaks, Melt [Bond and McKinley, 2008], LeakSurvivor [Tang et al., 2008], and Panacea [Breitgand et al., 2007] attempt to tolerate memory leaks at runtime. The shared goal of these tools is to extend the lifetime of applications with memory leaks.

6.1.1 Melt

Bond and McKinley [2008] introduce Melt, a memory leak tolerance tool that modifies the garbage collector and JIT compiler in the Jikes RVM. Melt marks the “staleness” of an object during that last garbage collection. An object is considered stale if it has not been accessed between subsequent garbage collections. Stale objects are stored in a stale space on disk. References from the live space to the stale space are managed using stub-scion pairs. Once a stale object is accessed, it is restored back to the live space through a swizzling process. Melt can execute the SPECjbb2000 benchmark approximately twice as long as without¹.

¹As measured by the total number of iterations.

Melt approximates leaks through staleness, making it subject to Type II errors—failing to detect a memory leak when there really is one. As each object’s liveness is dictated by the frequency of access, those objects which have been leaked, but continue to be accessed remain in the live space. For example, objects leaked in hash-based collections are subject to frequent re-hashing, preventing their detection as stale. [Bond and McKinley \[2008\]](#) refer to these type of leaks as “live” leaks. On the other hand, Melt can also be subject to Type I errors—incorrectly identifying objects as leaked. This condition is exemplified by the results of the SPECjbb2000 [[Standard Performance Evaluation Corporation, 2009](#)] benchmark. When Melt operates on this benchmark, it incorrectly identifies many non-leaked objects and moves them to the stale space, consequently incurring “significant activation overhead.” [[Bond and McKinley, 2008](#)]

6.1.2 LeakSurvivor

Developed in parallel, but independently from Melt, LeakSurvivor [[Tang et al., 2008](#)] similarly attempts to identify “potentially leaked” objects and move them to disk. Implemented in the Jikes RVM, LeakSurvivor uses Sleigh [[Bond and McKinley, 2006](#)] to identify stale objects. Sleigh uses Bit-Encoding Leak Location (Bell) to record the recency of object access. Bell minimizes spatial overhead using a statistical encoding process and encodes this information in four bits in each object header. The cost of this space efficiency is the decoding phase, which LeakSurvivor runs in an offline mode. In much the same way as Melt’s use of stub-scion pairs, LeakSurvivor uses a swap-out table to manage references from the live objects to potentially leaked objects.

Employing Sleigh analysis as an offline process², LeakSurvivor extends the

²Offline analysis entails preprocessing the benchmark with the Sleigh tool to detect potential

lifetime of the SPECjbb2000 benchmark by approximately a factor of 20³ and increasing the throughput by approximately 46%. The runtime performance of LeakSurvivor is sensitive to the rate of Type I errors. Sleight can avoid Type I errors with 99% confidence, at the cost of longer encoding times. In general, LeakSurvivor imposes more performance overhead for stale object identification than Melt, 21% compared to 6% respectively [Bond and McKinley, 2008]. As Sleight lacks support for moving objects, it may not use a copying garbage collector. Just as Melt is unable to identify “live leaks,” LeakSurvivor fails to detect what Tang et al. [2008] refer to as “semantic” memory leaks.

6.1.3 PANACEA

Breitgand et al. [2007] introduces Panacea, a self-healing framework that uses annotations to markup objects with healing hints. The healing annotations guide the various healing agents in performing their healing operations. The ObjectDump healer of Panacea dumps objects from main memory to auxiliary memory, retrieving them based on the demand of the system. The system designer associates `@Dumpable` annotation with each class the ObjectDump healer may manage. Each class marked as `@Dumpable` must be fully serializable. The ObjectDump healer dynamically generates proxy objects that get substituted for dumpable objects. The proxy object intercepts method invocations and swaps in the object from auxiliary memory if necessary before proxying to the dumpable object.

Generation of proxy objects is accomplished using Java dynamic proxy mechanism; however, this mechanism prohibits monitoring access to public fields of memory leak sources.

³As measured in total execution time, which fails to take into account the cost of the additional overhead.

objects. Panacea incurs a 15% overhead for object creation. While [Breitgand et al. \[2007\]](#) argue this cost is amortized for the lifetime of objects, [Bond and McKinley \[2008\]](#) argue that Panacea fails to scale for situations in which many small objects are leak culprits, as the vast majority of objects have short life spans. Using objects with size 10 MB, Panacea was able to instantiate four times as many objects (100 versus 25) within a heap size of 256 MB. However, access times to objects evicted to auxiliary memory were significantly (and non-linearly) reduced as the frequency and number of object instantiated increased.

While annotations provide the system designer with the ability to configure the healing process, doing so requires alteration to the source code. For existing systems, such alterations may be infeasible; on the other hand, instrumenting third-party libraries in their binary form becomes impossible.

6.1.4 Discussion

Memory leak tools are designed to tolerate leaks in systems that can be allocated sufficient memory to begin with. For example, the benchmarks performed against Melt used a heap size twice as large as required by the benchmark [[Bond and McKinley, 2008](#)]. JDiet is aimed at systems lacking memory sufficient to perform the desired operation from the start.

Even with the use of memory leak tolerance tools, applications subject to memory leaks will eventually exhaust the resources of the machine, albeit auxiliary memory rather than main memory. Melt does not collect the stale space and thus can leak disk space for stale objects that become unreachable after being moved to disk. LeakSurvivor does not reclaim disk space previously occupied by leaked objects. Similarly, JDiet performs no reclamation of the `Store` and

consequently leaks disk space for **Fat** objects reclaimed by the garbage collector. As [Tang et al. \[2008\]](#) indicate, this reduces the number of disk writes, favoring disk capacity over disk I/O.

While Melt and LeakSurvivor are solely aimed at memory leak tolerance, Panacea provides a suite of healer agents that can perform various autonomic healing activities. Panacea attempts to provide an implementation of autonomic computing, an area of research devoted to managing the rapidly growing complexity of software systems [[Sterritt and Hinchey, 2005](#)]. Other autonomic computing techniques include software rejuvenation [[Huang et al., 2004](#); [Kolettis and Fulton, 1995](#); [Vaidyanathan et al., 2001](#)], micro-rebooting [[Candea et al., 2002, 2004](#)], cloneable virtual machines [[Kawachiya et al., 2007](#)], and checkpointing [[Amza et al., 2000](#); [Chen et al., 1997](#); [Qin et al., 2007](#)].

Melt and LeakSurvivor are implemented in the Jikes RVM and require modifications to the garbage collector and other aspects of the runtime environment. On the other hand, Panacea operates at the library level, requires no modifications to the Java virtual machine. JDiet provides the benefits of both virtual machine- and library-based approaches; optimizing performance using bytecode engineering while avoiding virtual machine modification.

Panacea uses explicit annotations to configure the healing agents, which [Breitgand et al. \[2007\]](#) argue enhances readability, accessibility and maintainability of the application code. The ObjectDump healer supports the ability to specify the set of objects on which healing is performed as well as the paging policy. On the other hand, Melt and LeakSurvivor lack the ability to be configured on a case by case basis.

As exemplified by the SPECjbb2000 benchmark in Melt, leak tolerance tools

are often unable to “differentiate useful from useless memory accesses.” [Bond and McKinley, 2008]. So called “semantic” memory leaks are those which have objects which are reachable and frequently accessed, but loitering. This epitomizes a key feature of JDiet in contrast to existing approaches—the system designer is the decision maker. Instead of attempting to use heuristics to make guesses as to what data is being leaked, JDiet enables the system designer to make the decision as to what data can be evicted. Whereas Melt and LeakSurvivor suffer from Type I and II errors, JDiet does not suffer from such misclassification. The system designer, not JDiet is ultimately responsible for these decisions. Misclassification in other tools can degrade performance unnecessarily. Type I errors increase access times for non-leaked objects because they were incorrectly identified as leaked objects. On the other hand, Type II errors reduce the footprint reduction capabilities, as this condition occurs when objects that have been leaked are not evicted to auxiliary memory and unnecessarily occupy main memory.

6.2 Databases

Manually building a persistence layer with a common Database Management System (DBMS) requires the use of a query language supported by the DBMS. Most Relational Database Management Systems (RDBM) use the Structured Query Language (SQL). This approach has several notable drawbacks such as legibility, portability, readability and lack of reuse [Ortín et al., 2004]. Connector technologies such as Java Database Connectivity (JDBC) or Open Database Connectivity (ODBC) ease some of these drawbacks and DBMSs vendors supporting these connectors are ubiquitous.

Traditional SQL-based approaches introduce a dissonance between the rela-

tional structure of relational DBMSs and the object-oriented structure of object-oriented languages. This dissonance is commonly referred to as the “impedance mismatch” [Bauer and King, 2006; Russell, 2008; Zyl et al., 2006]. Complex object models, in particular, are not well suited for persistence in a traditional relational database management system [Brown and Morrison, 1992]. Programmers developing SQL-based querying solutions must have a thorough understanding of the mapping between the disparate object and relational structures. Object-relational mapping (ORM) tools alleviate this cognitive burden by providing a bridge between object-oriented data and relational database⁴. However, even with the use of an ORM, the system designer is required to understand the mapping process [Russell, 2008] and effects of caching [Keith and Stafford, 2008]. The Java Persistence API (JPA) provides architectural baseline for Java-based ORMs. Several implementations of this specification exist, notably Hibernate [Red Hat Middleware, 2009], Enterprise JavaBeans (EJB) [Sun Microsystems, Inc., 2009a], Java Data Objects (JDO) [Sun Microsystems, Inc., 2009b], and several others [Apache Software Foundation, 2009a,b; DataNucleus, 2009; Oracle, 2009c].

Object-oriented database management systems (OODBMS) [Prabhu, 2004] can also be used to reduce the impedance mismatch problem. Object-oriented databases, such as ObjectStore [Lamb et al., 1991] are engineered to optimize storage of complex objects, which are often ill-suited for storage in a relational database [Brown and Morrison, 1992]. However, some mapping between Java types and database types may still be required. Some applications are better suited for an OODBMS rather than a RDBMS and vice versa, depending on the complexity of the persistent object model. Zyl et al. [2006] provides a performance comparison between an OODBMS and a RDBMS with an ORM, while Zand

⁴ORM is being used in dynamic languages as well, such as Groovy [Richardson, 2008]

et al. [1995] provides a survey of object-oriented databases. Variations of object-oriented databases, such as SHORE [Carey et al., 1994] and QuickStore White and DeWitt [1995] use virtual memory mapping techniques

6.3 Orthogonal Persistence

Conventional methods for long-term data persistence are often complex, demanding and error prone [Zilio et al., 2001]. The database application programmer is often burdened with intricacies and manual interaction with the DBMS [Bläser, 2006], despite the automation provided by object-relational mapping and object-oriented databases. Many data intensive applications involve complex objects that are unsuited to conventional databases [Brown and Morrison, 1992]. Orthogonal persistence has been proposed as a means to alleviate these concerns.

Orthogonal persistence is predicated on the assertion that the longevity of data should be independent of the manner in which it is modified [Atkinson et al., 1983; Atkinson and Morrison, 1985]. Furthermore, persistence is orthogonal to an object's type [Atkinson and Morrison, 1995]. This means that any type in an orthogonal persistent system should be able to persist [Cooper and Wise, 1996]. The four core principles of orthogonal persistence include [Atkinson, 2001; Atkinson and Morrison, 1995]:

- **Type Orthogonality** requires that all objects may be made persistent, irrespective of their types. Type orthogonality proposes that the programmer should not be required to manually add information or capability to types to make them available for persistence. Data type orthogonality allows the modeling of data (object model) to be independent on the persistence of that data.

- **Transitivity** requires that persisting an object persists all encapsulated data by reachability. Whenever a persistent object is stored, all other non-transient objects and data it encapsulates must be persisted to ensure consistency. The lifetime of an object is based on reachability in the object graph. This principle is often referred to as “persistence by reachability.” [Atkinson et al., 1983]
- **Persistence Independence** requires the source need not be modified to facilitate persistence. A program with orthogonal persistence should look no different than an application without persistence [Moss and Hosking, 1996]. Persistence independence allows developers to devote less time to persistence code and focus more on application logic [Al-Mansari et al., 2007; Atkinson, 2001].
- **Persistent Identification** requires that the means for identification is orthogonal to the system using persistence. The identification for persistent entities must be managed by the persistence mechanism, not the objects themselves. Adherence to this principle upholds persistence independence.

Much research has been devoted to the discussion of the architectural principles of orthogonal persistence [Al-Mansari et al., 2007; Atkinson et al., 1983, 1996; Atkinson and Morrison, 1995; Cooper and Wise, 1996; Hosking and Chen, 1999a; Rashid and Chitchyan, 2003; Zigman and Blackburn, 1999]. [Cooper and Wise, 1996] refer to the above principles as “unrestricted” orthogonal persistence, while [Filman and Friedman, 2005] has termed these characteristics as persistence “obliviousness.”

6.3.1 Persistent Store

Orthogonally persistent systems are constructed such that they are bound to a persistent store [Atkinson and Morrison, 1995; Hosking and Chen, 1999a]. A persistent store is a dynamically allocated heap that persists from one execution to another [Appel and Li, 1991]. This persistent store is automatically and transparently synchronized with the persistent entities in the object graph. The persistent store can be viewed as an abstraction over disk memory and volatile memory [Cutts and Hosking, 1997]. The developer is not required to write objects to disk or flatten complex object models [Cooper and Wise, 1996]. Objects are stored and retrieved from the persistent store as needed automatically, and cached for performance volatile memory [Cutts and Hosking, 1997]. Consequently, orthogonal persistence demand three underlying components: (1) a persistent store (2) a residency check triggering and (3) data modification checks.

6.3.2 Implementation Approaches

6.3.2.1 Operating System Support

A simple means by which orthogonal persistence can be added to a programming language is running a program in that language within an operating system supporting persistence natively. [Atkinson and Morrison, 1995] describes such systems as “persistent worlds.” [Dearle et al., 1996] have made Java persistent by running it within the Grasshopper persistent operating system. However, some argue that the benefit of incorporating persistence into the operating system is dependent on the size of the application and length of the time period over which they run [Atkinson and Morrison, 1995]. As the program acquires persistence capability from the host operating system, this approach leads to lack of portability

between different operating environments.

6.3.2.2 Persistent Programming Languages

Unlike operating system support, persistent programming languages provide orthogonal persistence by incorporating persistence primitives and functionality directly into the programming language. It has been hypothesized that doing so requires only minimal changes to the language [Atkinson et al., 1983]. Unlike other approaches, persistent programming languages do not rely on the explicit use of a file system or database for long-term persistence. [Atkinson and Morrison, 1995] categorizes this approach to orthogonal persistence as an “integrated design,” as they combine features of programming languages and databases. Many persistent languages are built from existing non-persistent languages through significant alterations to the runtime system.

From the point of view of the programmer, persistent programming languages provide the illusion of a persistent heap [Hosking, 1991]. This heap is backed by a disk-based persistent store. At some point in time during the execution of a program, the contents of the heap are flushed to the persistent store, which [Hosking and Chen, 1999a] refers to as stabilization. During this process, all modified persistent objects are flushed to disk and non-persistent objects reachable from any persistent object are transitively persisted. Stabilization may happen at periodic checkpoints, during memory writes, or upon program termination.

The first notable persistent programming language was PS-Algol, a variant of Algol supporting a persistent heap [Atkinson et al., 1982]. A similar approach was taken with Smalltalk, which supports storing the heap on disk [Hosking, 1991]. However, these approaches are limited in that they only support as much

persistent data as can fit in the heap in main memory. To address scalability limitations, the Persistent Object Store supports the Napier88 persistent programming language [Brown and Morrison, 1992]. A similar is taken by ObjectStore, whose goal is to provide a unified view of data, both transient and persistent, while supporting common DBMS features such as transactions and versioning [Lamb et al., 1991]. ObjectStore is based on a memory mapped file input-output architecture, which provides the architectural basis for the INADA persistent programming language as well [Aritsugi and Makinouchi, 1995]. INADA supports a persistent pointer variable which identifies persistent data to be stored in the persistent heap. QuickStore is another memory mapped persistence mechanism [White and DeWitt, 1995], built on top of the EXODUS storage manager Carey et al. [1989].

The persistent stores managed in the memory mapped solutions are typically built on virtual memory. This approach uses virtual memory as a trigger for determining when persistent objects are moved from disk to main memory, consequently avoiding residency checks required by other approaches [Wilson, 1991]. Known as pointer swizzling [Atkinson and Morrison, 1995; Hosking and Chen, 1999a,b; Kemper and Kossmann, 1995; Lamb et al., 1991; Moss, 1992; Wilson, 1991], virtual memory references can be dereferenced efficiently to access persistent objects on disk. As JDiet manages only primitive data within each class in the inheritance hierarchy, pointer swizzling is unnecessary.

PM3 is an extension of the Module-3 system programming language supporting orthogonal persistence based on managed memory. Built on the SHORE object repository [Carey et al., 1994] and a mostly-copying garbage collection technique, PM3 is considered one of the first reachability-based persistent programming languages [Hosking and Chen, 1999a,b]. Persistent Active Oberon is

based on Active Oberon and uses modules as the fundamental unit of persistent [Bläser, 2006]. Fibonacci is an object database programming language with implicit persistence, querying, transaction and constraint support [Albano et al., 1995].

Dearle et al. [1996] argue that persistence support does not belong at the programming language level, but rather within the operating system. Because persistent programming languages are typically running within an operating system not constructed with persistence in mind, loss in efficiency and complexity often result. Most persistent programming languages require a persistence layer built on top of the operating system to abstract storage and retrieval, often duplicating parts of the functionality provided by the operating system.

6.3.2.3 Reflection

While persistent programming languages modify the static semantics of the language, reflection-based approaches alter the runtime semantics of language execution. Ortín et al. [2004] developed a persistence system built on nitrO [Ortín and Cueva, 2001, 2002; Ortín et al., 2003], a system supporting non-restrictive computational reflection. By dynamically modifying the language semantics at runtime, this persistence system automated persistent storage using one of two update policies: (1) simple, which persists an object after a certain number of access or (2) timed, which persists an object whenever a certain configurable time period elapses. A significant hindrance in the applicability of this approach is the runtime overhead imposed by nitrO.

6.3.2.4 Libraries

A less intrusive approach to orthogonal persistence is libraries. A library encapsulates a set of code engineered to automate persistence in a modular and reusable fashion. The prime example of this approach is orthogonal persistence for Java, otherwise known as PJava [Atkinson et al., 1996; Grimstad et al., 1999; Jordan and Atkinson, 1996, 1999]. PJava, developed by the Forest Project at Sun Microsystems Laboratories, like many other library and aspect-oriented approaches, uses the concept of persistent roots to associated objects to a persistent context [Atkinson et al., 1996]. At the heart of the PJava library is the class PJavaStore, which exposes the primary application programming interface. Persistent roots are registered explicitly with the PJavaStore class [Jordan and Atkinson, 1996], consequently violating persistence independence principle.

As with persistent programming languages, PJava is built on a disk-based persistent store, on top of which an object cache is placed which handles object faulting. As with many of the approaches to orthogonal persistence, a consolidated approach to handling transactions has yet to be sufficiently explored. However, Atkinson et al. [1996] suggests that transactions could be added to PJava with relative ease by employing transactions shells which are built on top of the runnable interface.

Library approaches, such as PJava, are predicated on the assumption that fewer lines of code required to incorporate persistence reduce maintenance costs and increase readability and comprehension [Grimstad et al., 1999]. In PJava, the introduction of persistence capabilities typically requires less than 100 lines of code, and is usually contained in very few areas in the code [Grimstad et al., 1999].

On the other hand, several limitations have prohibited PJava’s increased use. Early reports identified that concurrency issues in multi-thread systems using PJava could lead to loss of data integrity [Jordan and Atkinson, 1996]. Jordan and Atkinson [1999] has identified an issue appearing in the cross-session semantics of the hashCode implementations. By definition, the hash code implementation in a class is required to be consistent within only a single execution of the program. However, the use of PJava requires the hash code to be consistent across executions of the program.

Many third-party libraries and the core development kit do not frequently make use of the transient keyword, therefore causing data to be persistent by default [Jordan and Atkinson, 1996]. However, it is often the case that data in library-level objects is logically transient. Consequently, a significant pitfall of library approaches is their reliance on previously written and existing library to modify their syntactic structure to adhere to another library’s persistence approach [Al-Mansari et al., 2007]. Since the introduction of JOS in Java version 1.1, the use of the transient keyword has increased and might lessen the downside of transience issues [Jordan and Atkinson, 1999].

As a result of the coupled relationship between PJava and the underlying object semantics (hash code) and transience, Jordan and Atkinson [2000] put forth a Java Specification Request (JSR-20) in an effort to include orthogonal persistence concepts into the core language library for Java. However, this request was quickly rejected and has yet to gain support. Nettles and O’Toole [1993] introduce a similar system based on a copying garbage collector technique and implemented within a multi-threaded transaction management system. However, this approach makes an assumption that the persistent data set can be resident in main memory at all times.

A shortcoming of library-based approaches is their reliance on implicit management of persistent entities and lifecycles. While this prevents strict adherence to the principles of orthogonal persistence, some argue an implicit understanding of the persistence layer of an application is vital and such approaches still provide lucrative abstraction and reuse to programmers [Al-Mansari et al., 2007; Cooper and Wise, 1996; Hosking and Chen, 1999a; Rashid and Chitchyan, 2003].

6.3.2.5 Aspect-oriented

Aspect-oriented approaches attempt to reduce the implicit use of the persistence mechanism imposed by library-based approaches. Although persistence is often claimed to be a common cross-cutting concern, only a few systems leverage aspect-oriented technology for persistence with success, particularly those attempting to provide orthogonal persistence [Pawlak et al., 2004; Rashid and Chitchyan, 2003; Soares et al., 2002].

PersAJ provides a first step in developing a relationship between aspect-oriented approaches and persistence by storing aspects in an object-oriented database, in much the same way objects are stored [Rashid, 2001]. Rashid [2002] implements a means by which weaving persistence aspects could take place directly into an object-oriented database, by leveraging the SADES schema evolution functionality [Rashid et al., 2000]. Within the Java Aspect Components (JAC) system [Pawlak et al., 2004], persistence is aspectized along with distribution. However, this aspectization only encapsulates the concern of identifying persistent entities. Rashid and Chitchyan [2003] developed a persistence framework using several reusable aspects, including:

- *Connections* - the ability to connect and disconnect from a database

- *Storage and update* - the storage and modification of persistent objects
- *Deletion* - the removal of a persistent object from the store
- *Meta-data Access* - the access to table metadata used internally by other aspects, such as SQL translation
- *SQL Translation* - the translation mappings, providing an object-to-relational bridge

However, these aspects have several flaws. The storage and update aspects rely on strict encapsulation in persistent objects, as persistent data within an object is only accessed using accessor (get) and mutator (set) methods. This restriction is due to the reliance of the aspect on naming conventions for finding the approach accessor and mutator methods. Consequently, only publicly accessible data can be made persistent. The deletion aspect also uses a convention in that it advises all implementations of a common base class, `PersistentRoot`. The SQL translation, like many of the other components of this particular framework, rely heavily on reflection. Reflection, while useful for its generic self-introspection capabilities, usually comes with a heavy performance penalty. Lastly, the `PersistentRoot` class must be extended by any class requiring persistence support. In the current implementation, the `PersistentRoot` class is not an interface, leading to inheritance shortcomings [Al-Mansari et al., 2007], as Java does not support multiple inheritance. Al-Mansari et al. [2007] proposes persistence containers as a solution to the restrictive inheritance offered by most other systems, wherein objects added to the persistence container receive persistence capabilities.

In a parallel effort, Soares et al. [2002] developed similar persistence aspects while separating persistence and distribution concerns in a health care application. Using abstract pointcuts, these aspects rely on concrete aspects to specify

the precise join points in which persistence functionality is woven. Like previous approaches, these aspects often rely on naming conventions, marker annotations, or other explicit bindings with the persistence system. Despite the marginal gains acquired in terms of readability in comprehension [Soares et al. \[2002\]](#) note that synchronization and performance concerns in more complex systems limit the use of their approach.

Despite the various attempts to provide aspect-oriented orthogonal persistence, [Al-Mansari et al. \[2007\]](#) argue that none of these approaches fully support orthogonal persistence. In many of these solutions, the developer is still required to explicitly prepare types for persistence. Most aspect-oriented approaches rely on a persistent root class [[Cattell et al., 2000](#)] to distinguish between persistent and transient data [[Al-Mansari et al., 2007](#); [Rashid and Chitchyan, 2003](#)]. Therefore, a fully orthogonal, aspect-oriented persistent system has yet to be developed [[Rashid and Chitchyan, 2003](#)].

6.3.3 Discussion

The primary benefits of orthogonal persistence are reduced development complexity, increased productivity and reduced maintenance costs [[Al-Mansari et al., 2007](#); [Atkinson et al., 1996](#); [Atkinson and Morrison, 1995](#); [Cutts and Hosking, 1997](#)]. Persistence approaches, even those using object-relational mapping technologies, can incur heavy maintenance costs as developers must have a thorough understanding of the mappings between the code and the persistence mechanism [[Atkinson and Morrison, 1995](#)]. [Atkinson et al. \[1983\]](#) claim that systems constructed using orthogonal persistence are shorter in length, require less development time and have reduced maintenance costs compared to traditionally constructed systems. However, [Cooper and Wise \[1996\]](#) refute this claim and suggest

that the additional effort required to interface with modern non-orthogonally persistence systems is acceptable. The object model under orthogonal persistence, which does not require adherence to conventions (such as those in EJB), may also be more flexible and reusable, as the object model is not explicitly bound a particular persistence technology [Hosking and Chen, 1999a].

Proponents of orthogonal persistence claim additional software engineering benefits such as reusability and abstraction. Under the principles of orthogonal persistence, an application requires minimal or even no changes to its syntax in order for persistence support [Cutts and Hosking, 1997]. This method of persistence promotes abstraction and even transparency [Cutts and Hosking, 1997; Hosking and Chen, 1999a; Moss and Hosking, 1996].

Despite these claimed benefits, many argue against complete persistence obliviousness [Al-Mansari et al., 2007; Cooper and Wise, 1996; Hosking and Chen, 1999a; Rashid and Chitchyan, 2003]. They argue that some level of knowledge of about persistence should be made visible to the developer. For example, deletion of data from the persistent store must be explicitly considered [Hohenstein et al., 2007; Rashid and Chitchyan, 2003], often causing orthogonal persistence solutions to break the persistence independence principle. Common approaches, such as PJava [Atkinson et al., 1996], also break this rule because they employ the persistence-by-reachability concept, which requires explicit binding to a persistent root [Al-Mansari et al., 2007; Ortín et al., 2004; Rashid and Chitchyan, 2003]. Those approaches that require all persistent types to extend the persistent root type are subject to the fragile base class problem [Mikhajlov and Sekerinski, 1998]. There must also be a means by which transient data can be specified and excluded from automatic persistence [Moss and Hosking, 1996]. Bläser [2006] and Rashid and Chitchyan [2003] argue that a persistent programming language has

yet to be developed that adheres to the orthogonal persistence principles.

Complete transparency prohibits manual interaction with the database [Cutts and Hosking, 1997], including the benefits gained by traditional query languages such as SQL [Hohenstein et al., 2007; Rashid and Chitchyan, 2003]. Another traditional feature of most DBMSs lost by the orthogonal persistence approach is transactions [Atkinson et al., 1996]. Adherence to the principle of persistence independence precludes the developer from declaratively specifying the boundary points for transaction management [Al-Mansari et al., 2007]. The relationship between transactions and object destruction has also been subject to scrutiny [Zigman and Blackburn, 1999].

Performance problems may also be exposed by orthogonally persistent systems [Cutts and Hosking, 1997; Hosking and Chen, 1999a]. Persistence independent prohibits tuning techniques most database administrators are familiar with and often use to maximize performance for a particular application or environment. Cooper and Wise [1996] notes that systems built on non-orthogonal persistence approaches are much more efficient than those built with orthogonal persistence. Some techniques, such as pointer swizzling [Moss, 1992], can be employed to lessen the performance drawbacks. However, Rashid and Chitchyan [2003] note that there is a classic trade-off between genericity of the persistence solution and performance. Lastly, space saving techniques, such as compression, are much more difficult to implement within orthogonal persistence systems [Cooper and Wise, 1996].

6.4 External Memory Algorithms

External memory algorithms refer to the algorithms and data structures which perform explicit I/O management with auxiliary memory for disk-based computations [Vitter, 1998]. Kunkle and Cooperman [2008] employ clustering techniques to perform “disk-based computation” of Rubik’s cube puzzle solving at near RAM access speeds, demonstrating the viability of distributed computing for external memory algorithms. Vitter [2001] provides a survey of popular external memory algorithms and techniques.

6.5 Footprint Reduction, Compression & Optimization

Much research has been devoted to footprint reduction techniques, in a variety of approaches. A bulk of current research has been aimed at the embedded and mobile systems communities. Chen et al. [2003] provide an innovative Mark-Compress-Compact garbage collector that reduces the heap footprint using compression techniques during the compacting stage. Marinov and O’Callahan [2003] use object equality profiling to enable system designers to identify objects with potential equality. Zhang and Krintz [2005] present an adaptive code unloading algorithm to reduce memory footprint by dynamically unloading dead or infrequently used code. De Bosschere [2008] use binary rewriting techniques to minimize code while preserving program behavior and semantics. Pearce et al. [2007] provide a survey of code rewriting and optimization techniques targeted at embedded systems. Other footprint reduction techniques include heap compression [Bacon et al., 2002; Bonny and Henkel, 2007; Chen et al., 2005, 2003; Clausen et al., 2000; Lekatsas et al., 2000; Lekatsas and Wolf, 1999; Rizzo, 1997; Shaham et al., 2001; Venstermans et al., 2007], heap sharing [Choi and Han,

2008], bytecode optimization [Vallée-Rai et al., 1999, 2000], prolific types [Shuf et al., 2002], colocation [Yu et al., 2008] and many other optimization/reduction techniques [Ananian and Rinard, 2003; Guo et al., 2006; McDowell et al., 1998; Tip et al., 1999].

7 Conclusion

The JDiet footprint reduction tool is proposed as a means by which memory-constrained systems can combat limited main memory availability. Main memory scarcity is observed in many areas of software engineering and can be attributed to cost, size, or energy consumption of modern main memory. While several approaches can be taken to solve this problem, each suffer from restricting consequences. As such, there exists a need, particularly in the object-oriented development community, for a generic and configurable tool to reduce the runtime memory footprint of memory-constrained systems.

JDiet demonstrates innovative use of bytecode engineering techniques to balance the benefits and consequences of existing techniques. By leveraging the intuitive observations of *temporal cruciality* and *intra-object locality*, JDiet provides a more fine-grained eviction mechanism than previously offered. While naïve memory management techniques, such as virtual memory, evict pages, which may contain data for several objects, JDiet evicts to disk only the data specified by the system designer. As such, the system designer plays a vital role in providing JDiet with powerful application-specific intelligence regarding the cruciality of object attributes.

Guided by a sophisticated configuration system, JDiet employs aspect-oriented programming principles to decouple the memory management concern from the memory-constrained system. Using class-directional aspect coupling, JDiet is aware of the entities being managed, but not vice versa. Doing so offsets the costs that can be attributed to explicit persistence techniques, such as the use of an ORM on top of a RDBMS. Such approaches require that the programmer make preemptive, rather than reactive, decisions as to what data to evict

from main memory. This situation can lead to suboptimal decision making, as the decisions can be made inconsistently by different programmers, or can be made with a micro-level view of the system behavior. JDiet exploits knowledge of the runtime reference pattern of *fat objects* to allow eviction policies to make intelligent eviction decisions.

Extracting the concern of memory management from the memory-constrained system allows the system to be decoupled from the memory management solution. In systems that provide multiple deployment options, post-compile time configuration can be used to determine the degree to which JDiet manages the system. The system designer may provide two forms of the same system, a “light” version which utilizes the features of JDiet for running with a limited memory footprint and a “normal” version which does not utilize JDiet. The *configurability* and *portability* of JDiet permit such scaling configurations.

Benchmark evaluations demonstrate JDiet is capable of reducing the runtime memory footprint of an application performing DOM parsing by a maximum of 64.88% for a modest maximum heap size of 256 MB. As a result, a 225% larger input document was capable of being parsed within the same heap size. While adversely affecting the upfront performance cost of parsing (approximately 700%), the performance of iteration over the parsed document was increased for large input files, where garbage collection and thrashing become limiting performance factors. Performance benefits as large as 34% were observed for the depth-first iteration benchmark.

Despite the footprint reduction JDiet provides, the system designer must be aware of the tradeoffs involved when deciding what *fat attributes* are exposed to JDiet; memory reduction comes at the cost of performance. This epitomizes the class memory-performance tradeoff most software engineers face during design

and construction of software systems. *Fat attributes* should be selected such that the aggregate size of the *fat attributes* within a single class is much more than 20 bytes. As JDiet creates a *fat object* to store the *fat attributes*, doing so incurs a memory penalty of at least 20 bytes. The larger the *fat attributes*, the more memory footprint reduction JDiet achieves. Large primitive data types, particularly **Arrays** and **Strings** are ideal candidates for *fat attribute* selection.

While JDiet is applicable for extremely constrained systems, such as mobile computing and embedded systems, suiting the specific needs of this community is not the primary goal. JDiet aims to provide a generic solution that operates within the standard Java HotSpot virtual machine.

7.1 Future Work

7.1.1 Collections

Memory usage is not solely dependent on the size of the working data set, but rather the overhead imposed by data structures in which that data is organized [Chin et al., 2004; Pheng and Verbrugge, 2006; Rayside et al., 2006]. Mitchell and Sevitsky [2007] find that structural overhead of data structures can be significant. Currently, JDiet only supports the **Array** type as a repetitive data structure. As a result, an effort to support the built-in collection classes of the standard development kit (SDK) should be made. The Compile-time Weaver can be modified in order to support weaving the standard library classes, in a similar manner as Villazón et al. [2008].

7.1.2 Variability & Precision

To minimize variability within the benchmark evaluations, the replication count should be increased. The evaluation undertaken used a replication count of three, which is not sufficient for the detection of outliers. On only one occasion does this appear to be the case in the observed results.

Additionally, the metrics obtained during the benchmarking process were rough estimates, as many other confounding variables are involved. As a managed language, running benchmarks in Java is inherently subject to variations in output due to garbage collection behavior and interaction with the operating system.

7.1.3 Object-model Test Suite

The DOM benchmarks used during the evaluation of JDiet do not exhaustively test the potential use cases for JDiet. As such, the applicability of JDiet in situations beyond DOM parsing remains to be validated. To the knowledge of the author, there does not exist a comprehensive object-model benchmark that can be used as-is for validating JDiet, in particular, comparing against explicit persistence solutions such as Hibernate. As a result, future work on JDiet entail the development of an object-model benchmark, potentially based on Object-oriented Database Management System benchmarks (such as OO7 [[Carey et al., 1993](#)]). Such a suite of benchmarks would define a rich object-model, with complex relationships, data structures and inheritance.

7.1.4 Eviction Policies

Due to the limited nature of the evaluation undertaken for the prototype JDiet implementation, a rigorous comparison between the current eviction policies has yet to be made. In particular, an evaluation of the largest-object first (LOF) eviction policy is forthcoming. Furthermore, analysis of orphan **Fat** objects and their consequences on LOF will be made in the context of a more extensive evaluation.

Many other eviction policies can be implemented for use in JDiet, as the **EvictionPolicy** base classes provide for simple extension. Page-type oriented policies are ideal candidates for additional policies, as they can take into account object type. Future enhancements to JDiet could include the configuration of replacement policy on a per-type basis.

8 Appendices

A XMark Benchmark File Characteristics

The composition of the XMark [Schmidt et al., 2002] benchmarks used during evaluation is outlined in Table 3. While the size of each benchmark file increases, the relative composition of element and text nodes varies insignificantly.

Table 3: XMark benchmark file characteristics

Filename	Size (MB)	Nodes	% Element Nodes	% Text Nodes
auction_0.0.xml	0.034	1076	35.50	64.50
auction_0.025.xml	2.8	120026	35.51	64.49
auction_0.05.xml	5.5	235051	35.54	64.46
auction_0.075.xml	8.46	358147	35.53	64.47
auction_0.1.xml	14.54	472684	35.51	64.49
auction_0.2.xml	29.29	946556	35.52	64.48
auction_0.3.xml	43.67	1411836	35.52	64.48
auction_0.4.xml	58.13	1877979	35.53	64.47
auction_0.5.xml	72.21	2344425	35.53	64.47
auction_0.6.xml	87.57	2824179	35.53	64.47
auction_0.7.xml	102.26	3301416	35.52	64.48
auction_0.8.xml	116.44	3764891	35.52	64.48
auction_0.9.xml	131.01	4235340	35.53	64.47
auction_1.0.xml	144.96	4690647	35.52	64.48
auction_2.0.xml	290.74	9394467	35.53	64.47

B Configuration XML Schema

Listing 20 represents the XML Schema for the JDiet XML configuration files.

Listing 20: JDiet XML configuration schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns="http://www.csc.calpoly.edu/JDiet"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified">
  <xs:element name="Config">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Package">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="Class">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element name="Field">
                      <xs:complexType>
                        <xs:attribute name="name" type="xs:string"/>
                      </xs:complexType>
                    </xs:element>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

C Bytecode Example

Listing 21 displays the bytecode compiled for the `find` instance method of the `Buffer` class.

Listing 21: Example bytecode for a class method

```
aload_0
getfield <edu/calpoly/jdiet/buffer/Buffer.elements>
aload_1
invokeinterface <edu/calpoly/jdiet/fat/Fat.getID>
invokevirtual <gnu/trove/TIntObjectHashMap.contains>
ifne 62
getstatic <edu/calpoly/jdiet/buffer/Buffer.LOGGER>
ldc <Inserting new value: {}>
aload_1
invokeinterface <edu/calpoly/jdiet/fat/Fat.getID>
invokestatic <java/lang/Integer.valueOf>
invokeinterface <org/slf4j/Logger.debug>
aload_0
getfield <edu/calpoly/jdiet/buffer/Buffer.elements>
invokevirtual #166 <gnu/trove/TIntObjectHashMap.size>
aload_0
getfield <edu/calpoly/jdiet/buffer/Buffer.capacity>
if_icmpne 57
aload_0
invokevirtual <edu/calpoly/jdiet/buffer/Buffer.evict>
goto 57
astore_2
aload_0
aload_1
invokevirtual <edu/calpoly/jdiet/buffer/Buffer.buffer>
return
```

D Java Virtual Machine Instructions

The instruction set of Java virtual machine bytecode can be grouped into the following categories [[Sun Microsystems, Inc., 2006b](#)].

Loading and storing: Instructions that pop/push operands off/on the stack to/from local variables.

- *load*: `iload`, `iload_<n>`, `lload`, `lload_<n>`, `fload`, `fload_<n>`, `dload`, `dload_<n>`, `aload`, `aload_<n>`, `baload`, `caload`, `saload`, `iaload`, `laload`, `faload`, `daload`, `aaload`
- *store*: `istore`, `istore_<n>`, `lstore`, `lstore_<n>`, `fstore`, `fstore_<n>`, `dstore`, `dstore_<n>`, `astore`, `astore_<n>`, `bastore`, `castore`, `sastore`, `iastore`, `lastore`, `fastore`, `dastore`, `aastore`

Constant: Instructions to push operands on the stack from the runtime constant pool.

- `bipush`, `sipush`, `ldc`, `ldc_w`, `ldc2_w`, `aconst_null`, `iconst_m1`, `iconst_<i>`, `lconst_<l>`, `fconst_<f>`, `dconst_<d>`

Arithmetic: Instructions for performing arithmetic operations on the stack contents.

- *add*: `iadd`, `ladd`, `fadd`, `dadd`.
- *subtract*: `isub`, `lsub`, `fsub`, `dsub`.
- *multiply*: `imul`, `lmul`, `fmul`, `dmul`.
- *divide*: `idiv`, `ldiv`, `fdiv`, `ddiv`.
- *remainder*: `irem`, `lrem`, `frem`, `drem`.

- *negate*: ineg, lneg, fneg, dneg.
- *shift*: ishl, ishr, iushr, lshl, lshr, lushr.
- *bitwise or*: ior, lor.
- *bitwise and*: iand, land.
- *bitwise exclusive or*: ixor, lxor.
- *local variable increment*: iinc.
- *comparison*: dcmpg, dcmpl, fcmpg, fcml, lcmp.

Control flow: Instructions that alter the program counter, primarily under conditional values.

- *conditional branch*: feq, iflt, ifle, ifne, ifgt, ifge, ifnull, ifnonnull, if_icmpeq, if_icmpne, if_icmplt, if_icmpgt, if_icmple, if_icmpge, if_acmpeq, if_acmpn
- *compound conditional branch*: tableswitch, lookupswitch
- *unconditional branch*: goto, goto_w, jsr, jsr_w, ret

Type: Instructions that perform type operations that create objects, perform type checking or conversion.

- *construction*: new, newarray, anewarray, multianewarray
- *conversion*: i2l, i2f, i2d, l2f, l2d, f2d, i2b, i2c, i2s, l2i, f2i, f2l, d2i, d2l, and d2f

Class access: Instructions that provide access to the properties of an object.

- *arrays*: arraylength
- *fields*: getfield, putfield, getstatic, putstatic

Invocation: Instructions to perform method invocation and return values.

- `invokevirtual`, `invokeinterface`, `invokespecial`, `invokestatic`

Stack modification: Instructions that manipulate the stack contents.

- `pop`, `pop2`, `dup`, `dup2`, `dup_x1`, `dup2_x1`, `dup_x2`, `dup2_x2`, `swap`

Other: Instructions not falling in any of the above categories.

- *exceptions*: `athrow`
- *synchronization*: `monitorenter`, `monitorexit`

E UML Diagrams

Figure 16: Compile-time Weaver class design

Figure 17: Runtime Manager class design

F XMark Benchmark File Characteristics

The following appendices record the results from the evaluation reported in §5. Refer to Appendix F for a detailed description and summary of the content of each input file. A value of “-” indicates an `OutOfMemoryException` was thrown and the benchmark was unable to successfully complete.

F.1 32 MB Heap Size

Tables 4, 5, and 6 display the benchmark results when operating with a 32 MB maximum heap size. Table 4 summarizes the memory consumption for the XMark input files with and without JDiet in operation. The values reported are measured in megabytes (MB), rounded up to the nearest hundredth of a megabyte. Table 5 summarizes the total time taken to perform the parse of each input file, measure in seconds, rounded up to the nearest hundredth of a second. Table 6 summarizes the time taken to perform a depth-first iteration over each node in the tree parsed for each input file, measure in seconds, rounded up to the nearest hundredth of a second.

F.2 256 MB Heap Size

Tables 7, 8, and 9 display the benchmark results when operating with a 256 MB maximum heap size. Table 7 summarizes the memory consumption for the XMark input files with and without JDiet in operation. The values reported are measured in megabytes (MB), rounded up to the nearest hundredth of a megabyte. Table 8 summarizes the total time taken to perform the parse of each input file, measure in seconds, rounded up to the nearest hundredth of a second. Table 9 summarizes the time taken to perform a depth-first iteration over each

Table 4: Parse Memory Usage with 32 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	1.16	1.61	1.62	1.75
auction_0.025.xml	2.8	13.57	10.13	10.93	12.16
auction_0.05.xml	5.5	25.83	20.46	18.70	23.04
auction_0.075.xml	8.46	-	26.88	26.92	25.61
auction_0.1.xml	14.54	-	29.44	29.96	30.12
auction_0.2.xml	29.29	-	-	-	-
auction_0.3.xml	43.67	-	-	-	-
auction_0.4.xml	58.13	-	-	-	-
auction_0.5.xml	72.21	-	-	-	-
auction_0.6.xml	87.57	-	-	-	-
auction_0.7.xml	102.26	-	-	-	-
auction_0.8.xml	116.44	-	-	-	-
auction_0.9.xml	131.01	-	-	-	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

Table 5: Parse Total Time with 32 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	0.033	0.31	0.28	0.27
auction_0.025.xml	2.8	0.26	2.09	2.01	2.20
auction_0.05.xml	5.5	0.51	3.83	3.64	3.77
auction_0.075.xml	8.46	-	5.57	5.30	5.78
auction_0.1.xml	14.54	-	10.63	9.92	11.23
auction_0.2.xml	29.29	-	-	-	-
auction_0.3.xml	43.67	-	-	-	-
auction_0.4.xml	58.13	-	-	-	-
auction_0.5.xml	72.21	-	-	-	-
auction_0.6.xml	87.57	-	-	-	-
auction_0.7.xml	102.26	-	-	-	-
auction_0.8.xml	116.44	-	-	-	-
auction_0.9.xml	131.01	-	-	-	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

Table 6: Depth-first iteration Total Time with 32 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	0.01	0.01	0.01	0.01
auction_0.025.xml	2.8	0.06	0.06	0.05	0.05
auction_0.05.xml	5.5	0.11	0.10	0.10	0.10
auction_0.075.xml	8.46	-	0.15	0.15	0.15
auction_0.1.xml	14.54	-	0.19	0.28	0.20
auction_0.2.xml	29.29	-	-	-	-
auction_0.3.xml	43.67	-	-	-	-
auction_0.4.xml	58.13	-	-	-	-
auction_0.5.xml	72.21	-	-	-	-
auction_0.6.xml	87.57	-	-	-	-
auction_0.7.xml	102.26	-	-	-	-
auction_0.8.xml	116.44	-	-	-	-
auction_0.9.xml	131.01	-	-	-	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

node in the tree parsed for each input file, measure in seconds, rounded up to the nearest hundredth of a second.

F.3 1024 MB Heap Size

Tables 10, 11, and 12 display the benchmark results when operating with a 1024 MB maximum heap size. Table 10 summarizes the memory consumption for the XMark input files with and without JDiet in operation. The values reported are measured in megabytes (MB), rounded up to the nearest hundredth of a megabyte. Table 11 summarizes the total time taken to perform the parse of each input file, measure in seconds, rounded up to the nearest hundredth of a second. Table 12 summarizes the time taken to perform a depth-first iteration over each node in the tree parsed for each input file, measure in seconds, rounded

Table 7: Parse Memory Usage with 256 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	1.17	2.43	2.42	2.56
auction_0.025.xml	2.8	13.57	26.80	22.43	25.36
auction_0.05.xml	5.5	25.85	34.50	33.63	35.16
auction_0.075.xml	8.46	38.84	43.09	45.75	46.61
auction_0.1.xml	14.54	59.28	62.27	51.10	56.57
auction_0.2.xml	29.29	119.67	89.48	92.87	100.14
auction_0.3.xml	43.67	178.73	126.46	133.98	138.53
auction_0.4.xml	58.13	233.17	153.38	141.42	164.57
auction_0.5.xml	72.21	-	197.75	183.10	177.07
auction_0.6.xml	87.57	-	218.45	201.81	194.05
auction_0.7.xml	102.26	-	224.64	215.31	215.80
auction_0.8.xml	116.44	-	234.38	235.99	229.58
auction_0.9.xml	131.01	-	252.59	251.8	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

Table 8: Parse Total Time with 256 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	0.03	0.26	0.26	0.26
auction_0.025.xml	2.8	0.26	1.84	1.85	1.90
auction_0.05.xml	5.5	0.51	3.48	3.36	3.56
auction_0.075.xml	8.46	0.63	5.17	4.98	5.27
auction_0.1.xml	14.54	0.97	7.06	6.89	7.34
auction_0.2.xml	29.29	1.78	14.18	13.54	14.32
auction_0.3.xml	43.67	2.66	21.01	20.43	21.72
auction_0.4.xml	58.13	3.43	28.15	26.85	29.14
auction_0.5.xml	72.21	-	35.16	33.63	36.61
auction_0.6.xml	87.57	-	43.43	41.02	45.30
auction_0.7.xml	102.26	-	52.91	49.07	54.73
auction_0.8.xml	116.44	-	73.02	65.94	77.17
auction_0.9.xml	131.01	-	133.98	124.48	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

Table 9: Depth-first Iteration Total Time with 256 MB Max. Heap Size

Filename	File size	Without	With JDiet		
			FIFO	LOF	LRU
auction_0.0.xml	0.034	0.01	0.01	0.01	0.01
auction_0.025.xml	2.8	0.06	0.05	0.05	0.06
auction_0.05.xml	5.5	0.11	0.10	0.10	0.11
auction_0.075.xml	8.46	0.16	0.15	0.15	0.15
auction_0.1.xml	14.54	0.21	0.20	0.19	0.19
auction_0.2.xml	29.29	0.41	0.40	0.40	0.38
auction_0.3.xml	43.67	0.60	0.57	0.57	0.58
auction_0.4.xml	58.13	1.01	0.75	0.75	0.75
auction_0.5.xml	72.21	-	0.94	0.94	0.94
auction_0.6.xml	87.57	-	1.13	1.134	1.13
auction_0.7.xml	102.26	-	1.32	1.32	1.40
auction_0.8.xml	116.44	-	1.51	1.50	1.51
auction_0.9.xml	131.01	-	1.71	1.69	-
auction_1.0.xml	144.96	-	-	-	-
auction_2.0.xml	290.74	-	-	-	-

up to the nearest hundredth of a second.

Table 10: Parse Memory Usage with 1024 MB Max. Heap Size

			With JDiet		
Filename	File size	Without	FIFO	LOF	LRU
auction_0.0.xml	0.034	1.17	4.78	4.55	4.25
auction_0.025.xml	2.8	13.54	30.06	26.74	28.27
auction_0.05.xml	5.5	25.84	53.85	48.93	52.69
auction_0.075.xml	8.46	38.85	72.8	73.76	73.37
auction_0.1.xml	14.54	59.28	93.53	83.49	101.81
auction_0.2.xml	29.29	119.63	142.37	137.15	156.99
auction_0.3.xml	43.67	178.72	190.26	167.15	206.77
auction_0.4.xml	58.13	233.85	197.61	176.02	244.73
auction_0.5.xml	72.21	290.57	257.59	226.29	253.85
auction_0.6.xml	87.57	349.76	284.01	269.4	275.46
auction_0.7.xml	102.26	408.76	344.36	348.79	360.01
auction_0.8.xml	116.44	459.27	377.22	349.04	365.96
auction_0.9.xml	131.01	519.97	406.35	384.05	435.25
auction_1.0.xml	144.96	570.98	412.09	389.68	420.8
auction_2.0.xml	290.74	-	717.34	751.72	864.84

Table 11: Parse Total Time with 1024 MB Max. Heap Size

Filename	File size	Without	With JDiet		
			FIFO	LOF	LRU
auction_0.0.xml	0.034	0.05	0.28	0.40	0.32
auction_0.025.xml	2.8	0.29	1.66	1.63	1.74
auction_0.05.xml	5.5	0.53	2.95	2.85	3.22
auction_0.075.xml	8.46	0.72	4.71	4.45	5.28
auction_0.1.xml	14.54	1.14	6.72	6.41	7.52
auction_0.2.xml	29.29	1.96	13.57	13.07	16.18
auction_0.3.xml	43.67	3.02	20.61	19.82	25.67
auction_0.4.xml	58.13	4.18	28.21	26.53	36.52
auction_0.5.xml	72.21	4.69	34.68	33.28	49.10
auction_0.6.xml	87.57	5.04	41.59	40.52	65.70
auction_0.7.xml	102.26	7.5	49.01	47.35	76.70
auction_0.8.xml	116.44	7.63	55.67	53.79	57.61
auction_0.9.xml	131.01	7.95	62.88	60.55	64.94
auction_1.0.xml	144.96	9.28	70.45	67.24	71.93
auction_2.0.xml	290.74	-	141.47	136.10	144.85

Table 12: Depth-first Iteration Total Time with 1024 MB Max. Heap Size

Filename	File size	Without	With JDiet		
			FIFO	LOF	LRU
auction_0.0.xml	0.034	0.02	0.02	0.02	0.01
auction_0.025.xml	2.8	0.06	0.05	0.06	0.06
auction_0.05.xml	5.5	0.11	0.10	0.10	0.10
auction_0.075.xml	8.46	0.16	0.15	0.15	0.15
auction_0.1.xml	14.54	0.21	0.19	0.20	0.19
auction_0.2.xml	29.29	0.40	0.38	0.38	0.38
auction_0.3.xml	43.67	0.61	0.57	0.57	0.57
auction_0.4.xml	58.13	1.01	0.75	0.75	0.75
auction_0.5.xml	72.21	1.00	0.94	0.94	0.95
auction_0.6.xml	87.57	1.20	1.14	1.13	1.13
auction_0.7.xml	102.26	1.41	1.32	1.32	1.33
auction_0.8.xml	116.44	1.59	1.51	1.51	1.50
auction_0.9.xml	131.01	1.80	1.70	1.70	1.72
auction_1.0.xml	144.96	2.15	1.87	1.87	1.87
auction_2.0.xml	290.74	-	3.78	3.74	3.93

Glossary

fat class: A class representing the fat data elements extracted from a thin class.

fat object: An instance of a fat class.

first-in first-out (FIFO): Policy that evicts elements that were placed into the queue first.

footprint reduction: The process of reducing the memory footprint required by a system.

garbage collection: The automatic reclamation of computer storage.

hot-set: A set of frequently used pages in the query evaluation of a DBMS.

just-in-time (JIT) compiler: Optimization technique for interpreted languages that performs dynamic translation at runtime.

largest object first (LOF): Policy that evicts elements of the largest size first.

least-recently used (LRU): Policy that evicts elements that have been accessed least recently.

memory-constrained system: A system bounded by limited memory resources whose working set is larger than the main memory it has been allocated.

memory footprint: Maximum heap space consumed by a system throughout its execution.

Memory Management Unit (MMU): Operating system component responsible for managing virtual memory.

most-recently used (MRU): Policy that evicts elements that have been accessed most recently.

page hit: The situation in which a memory page being accessed is resident in main memory.

page miss: The situation in which a memory page being accessed is not resident in main memory and must therefore be swapped from auxiliary memory to main memory.

persistence: The period of time for which an object exists and is usable.

Persistent Application System (PAS): Application with long-lived, concurrently accessed, and potentially large bodies of data.

persistence durability: Principle of persistence that requires once a persistent *fat object* associated with an identity is stored, it can be subsequently retrieved by the same identity.

persistence detachment: Principle of persistence that requires once a persistent *fat object* associated with an identity is removed, it can no longer be retrieved by the same identity.

persistent programming language: A programming language that provides persistence primitives and functionality directly in the language.

reference locality: The natural tendency of a Database Management System to localize work within a set of database pages.

storage allocation problem: The problem of determining what data is resident in each part of the memory hierarchy at any time.

thin class: A class whose fat data elements have been extracted into a fat class.

thin object: An instance of a thin class.

virtual memory: Façade that provides an illusion of contiguous main memory where in fact such main memory may be fragmented or offloaded to auxiliary memory.

working set: The set of data a system can operate on within the memory allotted to it.

Index

- access modifier, [78](#)
- address space, [7](#)
- Ant, [84](#)
- ASM, [27](#), [73](#), [77](#)
- aspect, [29](#), [31](#)
- aspect-oriented, [4](#), [28–32](#), [43](#), [123](#), [125](#)
- aspect-oriented programming, [130](#)
- auxiliary memory, [3](#), [8](#), [9](#), [67](#)

- BCEL, *see* Bytecode Engineering Library
- BDB, *see* Berkeley Database
- Berkeley Database, [71](#), [78](#)
- buffer hit, [51](#)
- buffer manager, [9](#), [13–15](#)
- buffer miss, [52](#)
- buffer page, [14](#)
- buffer pool, [13](#)
- bytecode, [21](#), [26](#), [27](#), [138](#)
- Bytecode construction, [75](#)
- bytecode engineering, [27](#), [42](#), [43](#), [73](#)
- Bytecode Engineering Library, [27](#)
- bytecode manipulation, [84](#)
- Bytecode weaving, [75](#)

- CAD, *see* computer-aided design
- CAE, *see* computer-aided engineering
- CAM, *see* computer-aided manufacturing
- CASE, *see* computer-aided software engineering
- class file, [26](#)
- class-directional coupling, [130](#)
- clock algorithm, [17](#)
- Common Language Runtime, [10](#)
- Compile-time Weaver, [4](#), [43](#), [73](#), [76](#), [84](#), [85](#)
- computer-aided engineering, [34](#)
- computer-aided manufacturing, [34](#)
- computer-aided software engineering, [34](#)
- configuration, [84](#), [85](#), [136](#)
- CPU, *see* microprocessor
- cross-cutting concern, [28](#), [42](#), [123](#)
- CTW, *see* Compile-time Weaver, [76](#)

- dangling pointer, [11](#), [106](#)
- data layout, [13](#)
- data mining, [33](#)
- database, [9](#), [106](#)

Database Management System, [13–15](#), [18](#), [113](#), [127](#)
 DBMIN, [20](#)
 DBMS, *see* Database Management System, *see* Database Management System
 disk page, [9](#), [14](#)
 Document Object Model, [87](#), [88](#), [90](#)
 DOM, *see* Document Object Model
 domain separation, [19](#)
 double free, [106](#)
 double paging, [9](#), [15](#)
 drag, [107](#)
 dynamic proxy, [41](#)
 EJB, *see* Enterprise JavaBeans
 Enterprise JavaBeans, [114](#)
 eviction policy, [8](#), [9](#), [14](#), [16](#), [56](#)
 expert system, [33](#)
 extensible markup language, [75](#), [84](#), [85](#), [87](#), [88](#), [90](#), [136](#)
 fat class, [37](#), [38](#), [46](#), [151](#)
 fat extraction, [77](#), [78](#)
 fat object, [38](#), [47](#), [49](#), [50](#), [54–56](#), [60](#), [67](#), [151](#)
 FIFO, *see* first-in first-out
 finalization, [65](#)
 first-in first-out, [5](#), [18](#), [69](#), [151](#)
 footprint reduction, [3](#), [34](#), [36](#), [151](#)
 fragmentation, [9](#)
 functional locality, [28](#)
 garbage collection, [10–13](#), [22](#), [65](#), [67](#), [119](#), [151](#)
 garbage collector, [65](#)
 hash code, [22](#), [122](#)
 heap, [12](#), [23](#)
 heuristic, [56](#)
 hot-Set, [20](#)
 hot-set, [20](#)
 intra-object locality, [37](#), [130](#)
 J2SE, *see* Java 2 Standard Edition
 Java, [3](#), [10](#), [21](#), [22](#), [27](#), [36](#), [40](#), [41](#), [117](#), [121](#), [138](#)
 Java 2 Standard Edition, [53](#)
 Java Data Objects, [114](#)
 Java Database Connectivity, [26](#), [113](#)
 Java Object Serialization, [25](#), [26](#), [55](#), [71](#), [122](#)
 Java Runtime Environment, [53](#)
 Java virtual machine, [10](#), [21–23](#), [40](#), [79](#), [90](#), [112](#), [132](#), [138](#)
 JavaBeans, [25](#), [41](#)

Javassist, [27](#)
 JDBC, *see* Java Database Connectivity
 JDiet, [3](#), [4](#), [36–38](#), [42](#), [43](#), [54–57](#), [65](#),
 [73](#), [75](#), [76](#), [78](#), [84](#), [86](#), [88](#), [89](#),
 [92](#), [102–104](#), [111–113](#), [119](#), [130–](#)
 [134](#)
 JDO, *see* Java Data Objects
 JIT, *see* just-in-time compiler
 join point, [31](#)
 JOS, *see* Java Object Serialization
 JRE, *see* Java Runtime Environment
 just-in-time compiler, [13](#), [25](#), [151](#)
 JVM, *see* Java virtual machine

 K virtual machine, [1](#)
 KVM, *see* K virtual machine

 largest object first, [151](#)
 layer two cache, [72](#)
 least-frequently used, [68](#)
 least-recently used, [5](#), [17](#), [68](#), [69](#), [151](#)
 liveness, [107](#)
 load time weaving, [31](#)
 locality, [11](#), [13](#), [15](#), [19](#), [37](#)
 LOF, *see* largest object first
 logical page reference, [15](#)
 loitering, [107](#)

 lost pointer, [106](#)
 LRU, *see* least-recently used
 LTW, *see* load time weaving

 main memory, [1–3](#), [6](#), [7](#), [9](#), [49](#), [88](#)
 managed language, [10](#), [107](#)
 managed languages, [13](#)
 memory consumption, [86](#)
 memory footprint, [33](#), [151](#)
 memory leak, [10](#), [106](#), [107](#)
 memory management, [3](#), [7](#), [10](#)
 memory management unit, [8](#), [151](#)
 memory manager, [107](#)
 memory-constrained system, [1](#), [2](#), [4](#),
 [6](#), [33](#), [34](#), [130](#), [151](#)
 microprocessor, [6](#)
 MMU, *see* memory management unit
 monolocation, [49](#)
 most-recently used, [18](#), [19](#), [152](#)
 MRU, *see* most-recently used

 new algorithm, [19](#)

 object fault, [52](#)
 object swap, [50](#)
 object-oriented, [2](#), [21](#), [28](#), [29](#)
 object-oriented database, [14](#), [106](#)

- object-relational mapping, [106](#), [114](#),
[125](#), [130](#)
- obliviousness, [32](#)
- ODBC, *see* Open Database Connectivity
- ontology, [3](#)
- Open Database Connectivity, [113](#)
- operand stack, [24](#)
- operating system, [2](#), [8](#), [9](#), [14](#), [120](#)
- operating system, orthogonal persistence, [117](#)
- orphan fat object, [70](#)
- ORM, *see* object-relational mapping
- orphan, [67](#)
- Orphan fat object, [67](#)
- orthogonal persistence, [115–117](#), [122](#),
[125](#)
- page hit, [9](#), [152](#)
- page miss, [9](#), [16](#), [152](#)
- paging, [9](#)
- PAS, *see* Persistent Application System
- persistence, [106](#), [152](#)
- persistence detachment, [53](#)
- persistence durability, [53](#), [152](#)
- persistence independence, [3](#), [116](#)
- persistent application system, [33](#), [106](#),
[152](#)
- persistent heap, [118](#)
- persistent identification, [116](#)
- persistent programming language, [40](#),
[118](#), [120](#), [152](#)
- phantom reference, [22](#)
- physical memory, [7](#), [15](#), [23](#)
- physical page reference, [15](#)
- PJava, [3](#)
- plain old Java object, [41](#)
- POJO, *see* plain old Java object
- polymorphism, [21](#)
- QLSM, *see* query locality set model
- query locality set model, [20](#)
- RDBMS, *see* Relational Database Management System
- reachability, [11](#), [65](#), [107](#), [119](#)
- reference locality, [15](#), [16](#)
- reference queue, [67](#)
- reflection, [27](#), [40](#), [41](#), [120](#)
- Relational Database Management System, [113](#), [130](#)
- Remote Method Invocation, [25](#)
- resurrection, [65](#)
- RM, *see* Runtime Manager

RMI, *see* Remote Method Invocation
 runtime constant pool, 25
 Runtime Manager, 4, 43, 46, 54–56
 SAX, *see* Simple API for XML
 semantic equivalence, 46
 serializable, 55
 Simple API for XML, 87
 Simple Logging Façade for Java, 54
 SLF4J, *see* Simple Logging Façade
 for Java
 soft reference, 22
 SQL, *see* Structured Query Language
 stack, 24
 stack frame, 24
 static weaving, 31
 StAX, *see* Streaming API for XML
 storage allocation problem, 7, 152
 Streaming API for XML, 87
 strong reference, 67
 Structured Query Language, 113, 124, 127
 success, 86
 swap, 8, 23
 swizzling, 127
 symbolic reference, 25
 temporal cruciality, 36
 temporally cruciality, 130
 thin class, 38, 46, 152
 thin object, 38, 153
 thinning, 38
 thrashing, 9, 15, 23
 TIB, *see* type information block
 TLB, *see* translation lookaside buffer
 translation lookaside buffer, 8
 type checking, 22
 type descriptor, 26
 type information block, 21
 type orthogonality, 115
 UML, *see* Unified Modeling Language
 Unified Modeling Language, 5
 unmanaged language, 10
 unmanaged languages, 13
 unnecessary references, 107
 virtual machine, 22, 40
 virtual memory, 2, 7–9, 12, 14, 15, 23, 36, 119, 130, 153
 virtual method table, 21
 weak reference, 22, 65, 67
 weaving, 31
 working set, 13, 34, 36, 153
 Xerces, 87, 89

XML schema, [75](#), [136](#)

XStream, [85](#)

zombie reference, [107](#)

References

- AGESEN, O. AND GARTHWAITE, A. 2001. Efficient object sampling via weak references. *ACM SIGPLAN Notices* 36, 1, 121–126.
- AJWANI, D., DEMENTIEV, R., AND MEYER, U. 2006. A computational study of external-memory BFS algorithms. In *SODA '06: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*. ACM, New York, NY, USA, 601–610.
- AL-MANSARI, M., HANENBERG, S., AND UNLAND, R. 2007. Orthogonal persistence and AOP: a balancing act. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, New York, NY, USA, 2.
- ALBANO, A., GHELLI, G., AND ORSINI, R. 1995. Fibonacci: a programming language for object databases. *The VLDB Journal* 4, 3, 403–444.
- ALONSO, R. AND APPEL, A. W. 1990. An advisor for flexible working sets. *SIGMETRICS Performance Evaluation and Review* 18, 1, 153–162.
- AMZA, C., COX, A. L., AND ZWAENEPOEL, W. 2000. Data replication strategies for fault tolerance and availability on commodity clusters. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 459.
- ANANIAN, C. S. AND RINARD, M. 2003. Data size optimizations for Java programs. *SIGPLAN Not.* 38, 7, 59–68.
- ANDERSON, T. L. 1990. The HyperModel benchmark. In *Proceedings of the 2nd International Conference on Extending Database Technology*. Springer-Verlag New York, Inc., New York, NY, USA, 317–331.

- Apache Software Foundation 2005. *Xerces Java Parser*. Apache Software Foundation. <http://xerces.apache.org/xerces-j/>.
- Apache Software Foundation 2006. *BCEL*. Apache Software Foundation. <http://jakarta.apache.org/bcel/>.
- Apache Software Foundation 2007. *log4j*. Apache Software Foundation. <http://www.slf4j.org/>.
- Apache Software Foundation 2008a. *Apache Ant*. Apache Software Foundation. <http://ant.apache.org>.
- Apache Software Foundation 2008b. *Commons Logging*. Apache Software Foundation. <http://commons.apache.org/logging/>.
- Apache Software Foundation 2009a. *Cayenne*. Apache Software Foundation. <http://cayenne.apache.org/>.
- Apache Software Foundation 2009b. *OpenJPA*. Apache Software Foundation. <http://openjpa.apache.org/>.
- APPEL, A. W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letters* 25, 4, 275–279.
- APPEL, A. W. AND LI, K. 1991. Virtual memory primitives for user programs. *ACM SIGPLAN Notices* 26, 4, 96–107.
- ARGE, L. 1997. External-memory algorithms with applications in GIS. In *Algorithmic Foundations of Geographic Information Systems, this book originated from the CISM Advanced School on the Algorithmic Foundations of Geographic Information Systems*. Springer-Verlag, London, UK, 213–254.

- ARITSUGI, M. AND MAKINOCHI, A. 1995. Design and implementation of multiple type objects in a persistent programming language. In *Proceedings Nineteenth Annual International Computer Software and Applications Conference*. IEEE Press, Dallas, TX, USA, 70–76.
- ATKINSON, M., CHISHOLM, K., AND COCKSHOT, P. 1982. PS-algol: an algol with a persistent heap. *ACM SIGPLAN Notices* 17, 7, 24–31.
- ATKINSON, M. P. 2001. Persistence and java - a balancing act. In *Proceedings of the International Symposium on Objects and Databases*. Springer-Verlag, London, UK, 1–31.
- ATKINSON, M. P., BAILEY, P., CHISHOLM, K. J., COCKSHOT, P. W., AND R. MORRISON. 1983. An approach to persistent programming. *The Computer Journal* 26, 4 (Nov), 360–365.
- ATKINSON, M. P., DAYNÈS, L., JORDAN, M. J., PRINTEZIS, T., AND SPENCE, S. 1996. An orthogonally persistent Java. *ACM SIGMOD Record* 25, 4, 68–75.
- ATKINSON, M. P. AND MORRISON, R. 1985. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems* 7, 4, 539–559.
- ATKINSON, M. P. AND MORRISON, R. 1995. Orthogonally persistent object systems. *The VLDB Journal* 4, 3, 319–402.
- AZIMI, R., SOARES, L., STUMM, M., WALSH, T., AND BROWN, A. D. 2007. Path: page access tracking to improve memory management. In *Proceedings of the 6th International Symposium on Memory Management*. ACM, New York, NY, USA, 31–42.

- BACON, D. F., FINK, S. J., AND GROVE, D. 2002. Space- and time-efficient implementation of the Java object model. In *Proceedings of the 16th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science. Springer-Verlag, London, UK, 111–132.
- BARBOSA, D., MENDELZON, A., KEENLEYSIDE, J., AND LYONS, K. 2002. Toxgene: a template-based data generator for XML. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. ACM, New York, NY, USA, 616–616.
- BAUER, C. AND KING, G. 2006. *Java Persistence with Hibernate*. Manning Publications Co., Greenwich, CT, USA.
- BECERRA, Y., CORTES, T., GARCIA, J., AND NAVARRO, N. 2003. Evaluating the importance of virtual memory for Java. In *Proceedings of the 2003 IEEE International Symposium on Performance Analysis of Systems and Software*. IEEE Computer Society, Washington, DC, USA, 101–110.
- BENINI, L., MACII, A., AND PONCINO, M. 2003. Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques. *ACM Trans. Embed. Comput. Syst.* 2, 1, 5–32.
- BHADRA, S., CONRAD, A., HURKES, C., KIRKLIN, B., AND KAPFHAMMER, G. M. 2009. An empirical study of methods for executing test suites in memory constrained environments. In *Proceedings of the 4th Workshop on the Automation of Software Test*. Vancouver, Canada.
- BILARDI, G. AND PESERICO, E. 2001. A characterization of temporal locality and its portability across memory hierarchies. In *ICALP '01: Proceedings of*

- the 28th International Colloquium on Automata, Languages and Programming*, Springer-Verlag, London, UK, 128–139.
- BINDER, W., HULAAS, J., AND MORET, P. 2007. Advanced Java bytecode instrumentation. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*. ACM, New York, NY, USA, 135–144.
- BLACKBURN, S. M., GARNER, R., HOFFMANN, C., KHANG, A. M., MCKINLEY, K. S., BENTZUR, R., DIWAN, A., FEINBERG, D., FRAMPTON, D., GUYER, S. Z., HIRZEL, M., HOSKING, A., JUMP, M., LEE, H., MOSS, J. E. B., MOSS, B., PHANSALKAR, A., STEFANOVIĆ, D., VANDRUNEN, T., VON DINCKLAGE, D., AND WIEDERMANN, B. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. *ACM SIGPLAN Notices* 41, 10, 169–190.
- BLÄSER, L. 2006. A programming language with natural persistence. In *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, New York, NY, USA, 637–638.
- BOEHM, H.-J. 2003. Destructors, finalizers, and synchronization. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 262–272.
- BOND, M. D. AND MCKINLEY, K. S. 2006. Bell: bit-encoding online memory leak detection. *ACM SIGPLAN Notices* 41, 11, 61–72.
- BOND, M. D. AND MCKINLEY, K. S. 2008. Tolerating memory leaks. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Pro-*

- gramming Systems Languages and Applications*. ACM, New York, NY, USA, 109–126.
- BONNY, T. AND HENKEL, J. 2007. Instruction splitting for efficient code compression. In *DAC '07: Proceedings of the 44th annual conference on Design automation*. ACM, New York, NY, USA, 646–651.
- BRAUX, M. AND NOYÉ, J. 1999. Towards partially evaluating reflection in Java. In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation*. ACM, New York, NY, USA, 2–11.
- BRECHT, T., ARJOMANDI, E., LI, C., AND PHAM, H. 2006. Controlling garbage collection and heap growth to reduce the execution time of java applications. *ACM Transactions on Programming Languages and Systems* 28, 5, 908–941.
- BREITGAND, D., GOLDSTEIN, M., HENIS, E., AND SHEHORY, O. 2007. PANACEA - towards a self-healing development framework. In *Proceedings of the 10th IFIP/IEEE International Symposium on Integrated Network Management*. Munich, GER.
- BRICE, R. S. AND SHERMAN, S. W. 1977. An extension of the performance of a database manager in a virtual memory system using partially locked virtual buffers. *ACM Transactions on Database Systems* 2, 2, 196–207.
- BRODAL, G. S. AND FAGERBERG, R. 2003. On the limits of cache-obliviousness. In *STOC '03: Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM, New York, NY, USA, 307–315.
- BROWN, A. L. AND MORRISON, R. 1992. A generic persistent object store. *Software Engineering Journal* 7, 2 (Mar), 161–168.

- BRUNETON, E. 2007. ASM 3.0: A Java bytecode engineering library. <http://asm.ow2.org/doc/developer-guide.html>.
- CACHO, N., BATISTA, T., GARCIA, A., SANT'ANNA, C., AND BLAIR, G. 2006. Improving modularity of reflective middleware with aspect-oriented programming. In *Proceedings of the 6th international workshop on Software engineering and middleware*. ACM, New York, NY, USA, 31–38.
- CACHO, N., SANT'ANNA, C., FIGUEIREDO, E., GARCIA, A., BATISTA, T., AND LUCENA, C. 2006. Composing design patterns: a scalability study of aspect-oriented programming. In *Proceedings of the 5th international conference on Aspect-oriented software development*. ACM, New York, NY, USA, 109–121.
- CAI, F. F., HULL, M. E. C., AND BELL, D. A. 1997. Buffer management for high performance database systems. In *Proceedings of the High-Performance Computing on the Information Superhighway*. IEEE Computer Society, Washington, DC, USA, 633.
- CANDEA, G., CUTLER, J., FOX, A., DOSHI, R., GARG, P., AND GOWDA, R. 2002. Reducing recovery time in a small recursively restartable system. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 605–614.
- CANDEA, G., KAWAMOTO, S., FUJIKI, Y., FRIEDMAN, G., AND FOX, A. 2004. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation*. USENIX Association, Berkeley, CA, USA, 3–3.
- CAREY, M. J., DEWITT, D. J., FRANKLIN, M. J., HALL, N. E., MCAULIFFE,

- M. L., NAUGHTON, J. F., SCHUH, D. T., SOLOMON, M. H., TAN, C. K., TSATALOS, O. G., WHITE, S. J., AND ZWILLING, M. J. 1994. Shoring up persistent applications. *ACM SIGMOD Record* 23, 2, 383–394.
- CAREY, M. J., DEWITT, D. J., AND NAUGHTON, J. F. 1993. The OO7 benchmark. In *Proceedings of the ACM International Conference on Management of Data*. Washington, DC, USA, 12–21.
- CAREY, M. J., DEWITT, D. J., RICHARDSON, J. F., AND SHEKITA, E. J. 1989. Storage management for objects in EXODUS. In *Object-oriented concepts, databases, and applications*. ACM, New York, NY, USA, 341–369.
- CARR, R. W. AND HENNESSY, J. L. 1981. WSCLOCK—a simple and effective algorithm for virtual memory management. In *SOSP '81: Proceedings of the eighth ACM symposium on Operating systems principles*. ACM, New York, NY, USA, 87–95.
- CASAS, I. R. AND SEVCIK, K. C. 1989. A buffer management model for use in predicting overall database system performance. In *Proceedings of the Fifth International Conference on Data Engineering*. IEEE Computer Society, Washington, DC, USA, 463–469.
- CATTELL, R. G. G., BARRY, D., BERLER, M., EASTMAN, J., JORDAN, D., RUSSEL, C., SCHADOW, O., STENIENDA, T., AND VELEZ, F. 2000. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, San Francisco, CA, USA.
- CHEN, G., KANDEMIR, M., AND IRWIN, M. J. 2005. Exploiting frequent field values in Java objects for reducing heap memory requirements. In *Proceed-*

- ings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. ACM, New York, NY, USA, 68–78.
- CHEN, G., KANDEMIR, M., VIJAYKRISHNAN, N., IRWIN, M. J., MATHISKE, B., AND WOLCZKO, M. 2003. Heap compression for memory-constrained java environments. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 282–301.
- CHEN, K. AND CHEN, J.-B. 2007. Aspect-based instrumentation for locating memory leaks in Java programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference*. Vol. 2. IEEE Computer Society, Washington, DC, USA, 23–28.
- CHEN, Y., PLANK, J. S., AND LI, K. 1997. Clip: a checkpointing tool for message-passing parallel programs. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*. ACM, New York, NY, USA, 1–11.
- CHEW, K. AND SILBERSCHATZ, A. 1992. On the avoidance of the double paging anomaly in virtual memory systems. Tech. rep., Department of Computer Science, University of Texas at Austin, Austin, TX, USA.
- CHIANG, Y.-J., GOODRICH, M. T., GROVE, E. F., TAMASSIA, R., VENGROFF, D. E., AND VITTER, J. S. 1995. External-memory graph algorithms. In *SODA '95: Proceedings of the sixth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 139–149.
- CHIBA, S. 2000. Load-time structural reflection in java. In *ECOOP '00: Pro-*

- ceedings of the 14th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 313–336.
- CHILIMBI, T. M. AND LARUS, J. R. 1998. Using generational garbage collection to implement cache-conscious data placement. In *Proceedings of the 1st International Symposium on Memory Management*. ACM, New York, NY, USA, 37–48.
- CHIN, W.-N., CRACIUN, F., QIN, S., AND RINARD, M. 2004. Region inference for an object-oriented language. *ACM SIGPLAN Notices* 39, 6, 243–254.
- CHOI, J. P. 2000. Aspect-oriented programming with Enterprise JavaBeans. In *Proceedings of the 4th International Conference on Enterprise Distributed Object Computing*. IEEE Computer Society, Washington, DC, USA, 252.
- CHOI, Y. AND HAN, H. 2008. Shared heap management for memory-limited Java virtual machines. *Trans. on Embedded Computing Sys.* 7, 2, 1–32.
- CHOU, H.-T. AND DEWITT, D. J. 1985. An evaluation of buffer management strategies for relational database systems. In *Proceedings of the 11th International Conference on Very Large Data Bases*. VLDB Endowment, Stockholm, Sweden, 127–141.
- CLAUSEN, L. R., SCHULTZ, U. P., CONSEL, C., AND MULLER, G. 2000. Java bytecode compression for low-end embedded systems. *ACM Trans. Program. Lang. Syst.* 22, 3, 471–489.
- Codehaus 2008. *XStream*. Codehaus. <http://xstream.codehaus.org>.
- COLYER, A. AND CLEMENT, A. 2004. Large-scale AOSD for middleware. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA, 56–65.

- COOPER, T. AND WISE, M. 1996. Critique of orthogonal persistence. In *Proceedings of the Fifth International Workshop on Object-Orientation in Operating Systems*. IEEE Press, Seattle, WA, USA, 122–126.
- COURTS, R. 1988. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM* 31, 9, 1128–1138.
- CUTTS, Q. AND HOSKING, A. 1997. Analysing, profiling and optimising orthogonal persistence for Java. Tech. Rep. SMLI TR-97-63, Sun Microsystems Laboratories. Dec.
- DataNucleus 2009. *DataNucleus*. DataNucleus. <http://www.datanucleus.org/>.
- DE BOSSCHERE, K. 2008. Memory footprint reduction for embedded systems. In *SCOPES '08: Proceedings of the 11th international workshop on Software & compilers for embedded systems*. 31–31.
- DEARLE, A., HULSE, D., AND FARKAS, A. 1996. Persistent operating system support for Java. In *Proceedings of the 1st International Workshop on Persistence and Java*. Sun Microsystems, Inc., Drymen, Scotland.
- DENNING, P. J. 1968. The working set model for program behavior. *Communications of the ACM* 11, 5, 323–333.
- DENNING, P. J. 1970. Virtual memory. *ACM Computing Surveys* 2, 3, 153–189.
- DEVEDZIĆ, V. 2002. Understanding ontological engineering. *Communications of the ACM* 45, 4, 136–144.
- DINGLE, A. 2004. Reclaiming garbage and education: Java memory leaks. *Journal of Computing in Small Colleges* 20, 2, 8–16.

- DIWAN, A., TARDITI, D., AND MOSS, E. 1995. Memory system performance of programs with intensive heap allocation. *ACM Transactions on Computer Systems* 13, 3, 244–273.
- DONNELLY, K., HALLETT, J. J., AND KFOURY, A. 2006. Formal semantics of weak references. In *Proceedings of the 5th International Symposium on Memory Management*. ACM, New York, NY, USA, 126–137.
- EFFELSBURG, W. AND HAERDER, T. 1984. Principles of database buffer management. *ACM Transactions on Database Systems* 9, 4, 560–595.
- ej-technologies 2009. *JProfiler*. ej-technologies. <http://www.ej-technologies.com/products/jprofiler>.
- EKMAN, M. AND STENSTROM, P. 2004. A case for multi-level main memory. In *WMPI '04: Proceedings of the 3rd workshop on Memory performance issues*. ACM, New York, NY, USA, 1–8.
- ELRAD, T., FILMAN, R. E., AND BADER, A. 2001. Aspect-oriented programming: Introduction. *Communications of the ACM* 44, 10, 29–32.
- FALOUTSOS, C., NG, R., AND SELLIS, T. 1995. Flexible and adaptable buffer management techniques for database management systems. *IEEE Transactions on Computers* 44, 4 (Apr), 546–560.
- FERNANDEZ, E. B., LANG, T., AND WOOD, C. 1978. Effect of replacement algorithms on a paged buffer database system. *IBM Journal of Research and Development* 22, 2 (Mar), 185–196.
- FILHO, F. C., CACHO, N., FIGUEIREDO, E., RAQUEL MARANH A., GARCIA, A., AND RUBIRA, C. M. F. 2006. Exceptions and aspects: the devil is in the

- details. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, New York, NY, USA, 152–162.
- FILMAN, R. E. AND FRIEDMAN, D. P. 2005. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, R. E. Filman, T. Celrad, S. Clarke, and M. Aksit, Eds. Addison-Wesley, Boston, MA, 21–35.
- GABAY, Y. AND KFOURY, A. J. 2007. A calculus for Java’s reference objects. *ACM SIGPLAN Notices* 42, 8, 9–17.
- GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- GARCIA, A., SANT’ANNA, C., FIGUEIREDO, E., KULESZA, U., LUCENA, C., AND VON STAA, A. 2005. Modularizing design patterns with aspects: a quantitative study. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA, 3–14.
- GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. 2000. *Database system implementation*. Prentice Hall, Upper Saddle River, New Jersey.
- GNU Trove 2008. *GNU Trove: High Performance Collections for Java*. GNU Trove. <http://trove4j.sourceforge.net/>.
- GOETZ, B. 2004. *Java theory and practice: Garbage collection and performance*. IBM. <http://www.ibm.com/developerworks/library/j-jtp01274.html>.
- GOLDBERG, R. P. AND HASSINGER, R. 1974. The double paging anomaly. In *Proceedings of the AFIPS National Computer Conference*. Vol. 43. AFIPS Press, Montvale, N. J., 195–199.

- GOLDSTEIN, M., SHEHORY, O., AND WEINSBERG, Y. 2007. Can self-healing software cope with loitering? In *Proceedings of the Fourth International Workshop on Software Quality Assurance*. ACM, New York, NY, USA, 1–8.
- GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *The JavaTM Language Specification: Third Edition*. Addison-Wesley.
- GRAEFE, G. 2007. The five-minute rule twenty years later, and how flash memory changes the rules. In *DaMoN '07: Proceedings of the 3rd international workshop on Data management on new hardware*. ACM, New York, NY, USA, 1–9.
- GRIMSTAD, S., SJOBERG, D. I. K., ATKINSON, M. P., AND WELLAND, R. 1999. Evaluating usability aspects of pjama based on source code measurements. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 307–321.
- GRUNWALD, D., ZORN, B., AND HENDERSON, R. 1993. Improving the cache locality of memory allocation. *ACM SIGPLAN Notices* 28, 6, 177–186.
- GUO, Z., AMARAL, J. N., SZAFRON, D., AND WANG, Y. 2006. Utilizing field usage patterns for Java heap space optimization. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*. ACM, New York, NY, USA, 6.
- HALLETT, J. J. AND KFOURY, A. J. 2005. A formal semantics for weak references. Tech. Rep. Hal+Kfo:BUCS-TR-2005-X, Department of Computer Science, Boston University. May.

- HANNEMANN, J. AND KICZALES, G. 2002. Design pattern implementation in java and aspectj. *ACM SIGPLAN Notices* 37, 11, 161–173.
- HASTING, R. AND JOYCE, B. 1992. Purify: Fast detection of memory leaks and access errors. In *Winter USENIX Conference*. USENIX Association, San Francisco, CA, USA, 125–136.
- HAUSWIRTH, M. AND CHILIMBI, T. M. 2004. Low-overhead memory leak detection using adaptive statistical profiling. *ACM SIGPLAN Notices* 39, 11, 156–164.
- HAVERKORT, H., TOMA, L., AND ZHUANG, Y. 2009. Computing visibility on terrains in external memory. *Journal of Experimental Algorithms* 13, 1.5–1.23.
- HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.
- HERTZ, M. AND BERGER, E. D. 2005. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 313–326.
- HERTZ, M., FENG, Y., AND BERGER, E. D. 2005. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 143–153.
- HIRZEL, M. 2007. Data layouts for object-oriented programs. *SIGMETRICS Performance Evaluation and Review* 35, 1, 265–276.
- HIRZEL, M., DIWAN, A., AND HOSKING, A. L. 2001. On the usefulness of liveness for garbage collection and leak detection. In *Proceedings of the 15th*

- European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 181–206.
- HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. 2002. Understanding the connectivity of heap objects. In *Proceedings of the 3rd International Symposium on Memory Management*. ACM, New York, NY, USA, 36–49.
- HOHENSTEIN, U., MEUNIER, R., AND SCHWANNINGER, C. 2007. An aspect-oriented implementation of the ejb3.0 persistence concept. In *Proceedings of the 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM, New York, NY, USA, 4.
- HOSKING, A. L. 1991. Main memory management for persistence. In *The OOPSLA '91 Workshop on Garbage Collection*. ACM Press, Phoenix, AZ, USA.
- HOSKING, A. L. AND CHEN, J. 1999a. Mostly-copying reachability-based orthogonal persistence. *ACM SIGPLAN Notices* 34, 10, 382–398.
- HOSKING, A. L. AND CHEN, J. 1999b. PM3: An orthogonal persistent systems programming language - design, implementation, performance. In *Proceedings of the 25th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 587–598.
- HUANG, X., BLACKBURN, S. M., MCKINLEY, K. S., MOSS, J. E. B., WANG, Z., AND CHENG, P. 2004. The garbage collection advantage: improving program locality. *ACM SIGPLAN Notices* 39, 10, 69–80.
- JIANG, S., CHEN, F., AND ZHANG, X. 2005. Clock-pro: an effective improvement of the clock replacement. In *Proceedings of the Annual Conference on*

- USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 35–35.
- Jike RVM Project 2009. *Jikes RVM*. Jike RVM Project. <http://jikesrvm.org/>.
- JONES, R. E. AND RYDER, C. 2008. A study of java object demographics. In *Proceedings of the 7th International Symposium on Memory Management*. ACM, New York, NY, USA, 121–130.
- JORDAN, M. J. AND ATKINSON, M. P. 1996. Early experiences with persistent Java. In *Proceedings of the 1st International Workshop on Persistence and Java*. Sun Microsystems, Inc., Glasgow, Scotland.
- JORDAN, M. J. AND ATKINSON, M. P. 1999. Orthogonal persistence for java? - a mid-term report. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International Workshop on Persistence and Java (PJW3)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 335–352.
- JORDAN, M. J. AND ATKINSON, M. P. 2000. Orthogonal persistence for the JavaTM Platform: Specification and rationale. Tech. Rep. SMLI TR-2000-94, Sun Microsystems Laboratories, Mountain View, CA, USA.
- JUMP, M. AND MCKINLEY, K. S. 2007. Cork: dynamic memory leak detection for garbage-collected languages. *ACM SIGPLAN Notices* 42, 1, 31–38.
- KANTARDZIC, M. 2002. *Data Mining: Concepts, Models, Methods, and Algorithms*. Wiley-IEEE Press.
- KATZ, R. H. 1990. Toward a unified framework for version modeling in engineering databases. *ACM Computing Surveys* 22, 4, 375–409.

- KAWACHIYA, K., OGATA, K., AND ONODERA, T. 2008. Analysis and reduction of memory inefficiencies in Java strings. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*. ACM, New York, NY, USA, 385–402.
- KAWACHIYA, K., OGATA, K., SILVA, D., ONODERA, T., KOMATSU, H., AND NAKATANI, T. 2007. Cloneable JVM: a new approach to start isolated Java applications faster. In *Proceedings of the 3rd International Conference on Virtual Execution Environments*. ACM, New York, NY, USA, 1–11.
- KEITH, M. AND STAFFORD, R. 2008. Exposing the orm cache. *Queue* 6, 3, 38–47.
- KEMPER, A. AND KOSSMANN, D. 1995. Adaptable pointer swizzling strategies in object bases: design, realization, and quantitative analysis. *The VLDB Journal* 4, 3, 519–567.
- KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. G. 2001. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 327–353.
- KIENZLE, J. AND GUERRAOUI, R. 2002. AOP: Does it make sense? the case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 37–61.
- KITCHENHAM, B., LINKMAN, S., AND LAW, D. 1994. Critical review of quantitative assessment. *Software Engineering Journal* 9, 2 (Mar), 43–53.
- KNUTH, D. E. 1969. *Fundamental Algorithms*. Vol. 1 of The Art of Computer Programming. Addison-Wesley, Reading, MA.

- KOLETTIS, N. AND FULTON, N. D. 1995. Software rejuvenation: Analysis, module and applications. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*. IEEE Computer Society, Washington, DC, USA, 381.
- KULESZA, U., SANT'ANNA, C., AND LUCENA, C. 2005. Refactoring the JUnit framework using aspect-oriented programming. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 136–137.
- KUNKLE, D. AND COOPERMAN, G. 2008. Solving Rubik's Cube: disk is the new RAM. *Communications of the ACM* 51, 4, 31–33.
- KUROTSUCHI, B. T. 1997. The wonders of Java object serialization. *Crossroads* 4, 2, 3–8.
- LAMB, C., LANDIS, G., ORENSTEIN, J., AND WEINREB, D. 1991. The Object-Store database system. *Communications of the ACM* 34, 10, 50–63.
- LANG, T., WOOD, C., AND FERNÁNDEZ, E. B. 1977. Database buffer paging in virtual storage systems. *ACM Transactions on Database Systems* 2, 4, 339–351.
- LATTNER, C. AND ADVE, V. 2005. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, New York, NY, USA, 129–142.
- LEAL, M. A. AND IERUSALIMSKY, R. 2004. The weak-reference interface. Online.
- LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000. Code compression for low

- power embedded system design. In *DAC '00: Proceedings of the 37th conference on Design automation*. ACM, New York, NY, USA, 294–299.
- LEKATSAS, H. AND WOLF, W. 1999. Samc: A code compression algorithm for embedded processors. *IEEE Transactions on CAD* 18, 1689–1701.
- LEVY, H., MESSINGER, T., AND MORRIS, R. 1996. The cache assignment problem and its application to database buffer management. *IEEE Transactions on Software Engineering* 22, 11 (Nov), 827–838.
- LI, Y. G., BRESSAN, S., DOBBIE, G., LACROIX, Z., LEE, M. L., NAMBIAR, U., AND WADHWA, B. 2001. XOO7: applying OO7 benchmark to XML query processing tool. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*. ACM, New York, NY, USA, 167–174.
- LINDHOLM, T. AND YELLIN, F. 1999. The Java Virtual Machine Specification. Tech. Rep. Second Edition, Sun Microsystem, Inc.
- LIPPERT, M. AND LOPES, C. V. 2000. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM, New York, NY, USA, 418–427.
- LIU, B. 2008. *Web Data Mining: Exploring Hyprlinks, Contents, and Usage Data*. Springer-Verlag.
- LIU, H., ROEDER, T., WALSH, K., BARR, R., AND SIRER, E. G. 2005. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. ACM, New York, NY, USA, 149–162.

- MAHAPATRA, N. R. AND VENKATRAO, B. 1999. The processor-memory bottleneck: problems and solutions. *Crossroads* 5, 3es, 2.
- MARINOV, D. AND O'CALLAHAN, R. 2003. Object equality profiling. *ACM SIGPLAN Notices* 38, 11, 313–325.
- MARNEY, S. AND IBRAHIM, M. 1995. Using objects to manage in-memory data intensive expert systems. In *OOPSLA '95: Addendum to the proceedings of the 10th annual conference on Object-oriented programming systems, languages, and applications (Addendum)*. ACM, New York, NY, USA, 67–71.
- MCDOWELL, C. E., MONTAGUE, B. R., ALLEN, M. R., BALDWIN, E. A., AND MONTOREANO, M. E. 1998. Javacam: Trimming Java down to size. *IEEE Internet Computing* 2, 3, 53–59.
- MIKHAJLOV, L. AND SEKERINSKI, E. 1998. A study of the fragile base class problem. In *Proceedings of the 12th European Conference on Object-Oriented Programming*. Springer-Verlag, London, UK, 355–382.
- MITCHELL, N. AND SEVITSKY, G. 2003. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large Java applications. In *European Conference on Object-Oriented Programming*. 351–377.
- MITCHELL, N. AND SEVITSKY, G. 2007. The causes of bloat, the limits of health. *ACM SIGPLAN Notices* 42, 10, 245–260.
- MORRISETT, G., FELLEISEN, M., AND HARPER, R. 1995. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*. ACM Press, New York, NY, USA, 66–77.

- MOSS, J. E. B. 1992. Working with persistent objects: to swizzle or not to swizzle. *IEEE Transactions on Software Engineering* 18, 8, 657–673.
- MOSS, J. E. B. AND HOSKING, A. L. 1996. Approaches to adding persistence to java. Tech. Rep. SMLI TR-96-58, Sun Microsystems Laboratories, Drymen, Scotland. Sep.
- NETHERCOTE, N. AND SEWARD, J. 2007. Valgrind: A framework for heavy-weight dynamic binary instrumentation. *ACM SIGPLAN Notices* 42, 6, 89–100.
- NETTLES, S. AND O'TOOLE, J. 1993. Implementing orthogonal persistence: a simple optimization using replicating collection. In *Proceedings of the Third International Workshop on Object Orientation in Operating Systems*. IEEE Press, Asheville, NC, USA, 177–181.
- NGUYEN, H. H. AND RINARD, M. 2007. Detecting and eliminating memory leaks using cyclic memory allocation. In *Proceedings of the 6th international symposium on Memory management*. ACM, New York, NY, USA, 15–30.
- Object Management Group 2009. *UML*. Object Management Group. <http://www.uml.org/>.
- ObjectWeb 2007. *ASM*. ObjectWeb. <http://asm.ow2.org/>.
- O'HAIR, K. 2004. *HPROF: A heap/CPU profiling tool in J2SE 5.0*. Sun Microsystems. <http://java.sun.com/developer/technicalArticles/Programming/HPROF.html>.
- OLSON, M. A., BOSTIC, K., AND SELTZER, M. 1999. Berkeley db. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 43–43.

O'NEIL, E. J., O'NEIL, P. E., AND WEIKUM, G. 1993. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD Record* 22, 2, 297–306.

ORACLE. 2008. Getting started with Berkeley DB Java Edition. Tech. rep., Oracle. June.

ORACLE. 2009a. A comparison of Oracle Berkeley DB and relational database management systems. Tech. rep., Oracle. March.

Oracle 2009b. *Oracle Berkeley DB Java Edition*. Oracle.
<http://www.oracle.com/technology/products/berkeley-db/je/index.html>.

Oracle 2009c. *TopLink*. Oracle. <http://www.oracle.com/technology/products/ias/toplink/index.htm>.

ORTÍN, F. AND CUEVA, J. M. 2001. Building a completely adaptable reflective system. In *Workshop on Adaptive Object-Models and Metamodeling Techniques*. Campus Llamaquique, Budapest, Hungary, 1–9.

ORTÍN, F. AND CUEVA, J. M. 2002. The nitroO relective platform. In *International Conference on Software Engineering Research and Practice*. Las Vegas, NV, USA.

ORTIN, F., CUEVA, J. M., AND MARTINEZ, A. B. 2003. The reflective nitroO abstract machine. *SIGPLAN Not.* 38, 6, 40–49.

ORTÍN, F., LOPEZ, B., AND PEREZ-SCHOFIELD, J. B. G. 2004. Separating adaptable persistence attributes through computational reflection. *IEEE Software* 21, 6, 41–49.

PANDA, P. R., CATTHOOR, F., DUTT, N. D., DANCKAERT, K., BROCKMEYER, E., KULKARNI, C., VANDERCAPPELLE, A., AND KJELDSBERG,

- P. G. 2001. Data and memory optimization techniques for embedded systems. *ACM Trans. Des. Autom. Electron. Syst.* 6, 2, 149–206.
- PAPAPETROU, O. AND PAPADOPOULOS, G. A. 2004. Aspect oriented programming for a component-based real life application: a case study. In *Proceedings of the 2004 ACM Symposium on Applied Computing*. ACM, New York, NY, USA, 1554–1558.
- PAWLAK, R., SEINTURIER, L., DUCHIEN, L., FLORIN, G., LEGOND-AUBRY, F., AND MARTELLI, L. 2004. JAC: an aspect-based distributed dynamic framework. *Software Practice and Experience* 34, 12, 1119–1148.
- PEARCE, D. J., WEBSTER, M., BERRY, R., AND KELLY, P. H. J. 2007. Profiling with aspectj. *Software Practice and Experience* 37, 7, 747–777.
- PHENG, S. AND VERBRUGGE, C. 2006. Dynamic data structure analysis for Java programs. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*. IEEE Computer Society, Washington, DC, USA, 191–201.
- PHIPPS, G. 1999. Comparing observed bug and productivity rates for Java and C++. *Software Practice and Experience* 29, 4, 345–358.
- PRABHU, C. S. R. 2004. *Object Oriented Database Systems*. Prentice-Hall of India Pvt.Ltd.
- PRINTEZIS, T. 1996. Analysing a simple disk garbage collector. Technical Report TR-96-58, SunLabs.
- PULVERMUELLER, E., KLAEREN, H., AND SPECK, A. 1999. Aspects in distributed environments. In *Proceedings of the First International Symposium*

- on *Generative and Component-Based Software Engineering (GCSE)*. Lecture Notes in Computer Science, vol. 1799. Springer-Verlag, Erfurt, Germany, 37–48.
- QIN, F., LU, S., AND ZHOU, Y. 2005. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, USA, 291–302.
- QIN, F., TUCEK, J., ZHOU, Y., AND SUNDARESAN, J. 2007. Rx: Treating bugs as allergies—a safe method to survive software failures. *ACM Transactions on Computer Systems* 25, 3, 7.
- Quality Open Software 2009. *SLF4J*. Quality Open Software. <http://www.slf4j.org/>.
- RASHID, A. 1998. SADES - a semi-autonomous database evolution system. In *Workshop on Object-Oriented Technology*. Springer-Verlag, London, UK, 24–25.
- RASHID, A. 2001. On to aspect persistence. In *Proceedings of the Second International Symposium on Generative and Component-Based Software Engineering-Revised Papers*. Springer-Verlag, London, UK, 26–36.
- RASHID, A. 2002. Weaving aspects in a persistent environment. *ACM SIGPLAN Notices* 37, 2, 36–44.
- RASHID, A. AND CHITCHYAN, R. 2003. Persistence as an aspect. In *Proceedings of the 2nd International Conference on Aspect-oriented Software Development*. ACM, New York, NY, USA, 120–129.

- RASHID, A. AND LOUGHRAN, N. 2003. Relational database support for aspect-oriented programming. In *Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*. Springer-Verlag, London, UK, 233–247.
- RASHID, A. AND PULVERMUELLER, E. 2000. From object-oriented to aspect-oriented databases. In *Proceedings of the 11th International Conference on Database and Expert Systems Applications*. Springer-Verlag, London, UK, 125–134.
- RASHID, A. AND SAWYER, P. 2000. Object database evolution using separation of concerns. *ACM SIGMOD Record* 29, 4, 26–33.
- RASHID, A., SAWYER, P., AND PULVERMUELLER, E. 2000. A flexible approach for instance adaptation during class versioning. In *Proceedings of the International Symposium on Objects and Databases*. Lecture Notes in Computer Science, vol. 1944. Springer-Verlag, London, UK, 101–113.
- RAYSIDE, D. AND MENDEL, L. 2007. Object ownership profiling: a technique for finding and fixing memory leaks. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, New York, NY, USA, 194–203.
- RAYSIDE, D., MENDEL, L., AND JACKSON, D. 2006. A dynamic analysis for revealing object ownership and sharing. In *Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*. ACM, New York, NY, USA, 57–64.
- Red Hat Middleware 2009. *Hibernate - Relational Persistence for Java and .NET*. Red Hat Middleware. <https://www.hibernate.org/>.
- RICHARDSON, C. 2008. ORM in dynamic languages. *Queue* 6, 3, 28–37.

- RICHARDSON, J. AND SCHWARZ, P. 1991. Aspects: extending objects to support multiple, independent roles. *ACM SIGMOD Record* 20, 2, 298–307.
- RIZZO, L. 1997. A very fast algorithm for RAM compression. *SIGOPS Oper. Syst. Rev.* 31, 2, 36–45.
- RUSSELL, C. 2008. Bridging the object-relational divide. *Queue* 6, 3, 18–28.
- SACCO, G. M. AND SCHKOLNICK, M. 1982. A mechanism for managing the buffer pool in a relational database system using the hot set model. In *Proceedings of the 8th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 257–262.
- SACCO, G. M. AND SCHKOLNICK, M. 1986. Buffer management in relational database systems. *ACM Transactions on Database Systems* 11, 4, 473–498.
- SCHMIDT, A., WAAS, F., KERSTEN, M., CAREY, M. J., MANOLESCU, I., AND BUSSE, R. 2002. XMark: a benchmark for XML data management. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 974–985.
- SCHOEBERL, M. AND VITEK, J. 2007. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems*. ACM, New York, NY, USA, 85–93.
- SHAHAM, R., KOLODNER, E. K., AND SAGIV, S. 2001. Heap profiling for space-efficient java. *ACM SIGPLAN Notices* 36, 5 (May), 104–113.
- SHERMAN, S. W. AND BRICE, R. S. 1976a. I/O buffer performance in a virtual memory system. In *Proceedings of the 4th Symposium on Simulation of Computer Systems*. IEEE Press, Piscataway, NJ, USA, 25–35.

- SHERMAN, S. W. AND BRICE, R. S. 1976b. Performance of a database manager in a virtual memory system. *ACM Transactions on Database Systems* 1, 4, 317–343.
- SHIM, H., JOO, Y., CHOI, Y., LEE, H. G., AND CHANG, N. 2003. Low-energy off-chip sdram memory systems for embedded applications. *ACM Transactions on Embedded Computing Systems* 2, 1, 98–130.
- SHUF, Y., GUPTA, M., BORDAWEKAR, R., AND SINGH, J. P. 2002. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 295–306.
- SIEGWART, D. AND HIRZEL, M. 2006. Improving locality with parallel hierarchical copying gc. In *Proceedings of the 5th International Symposium on Memory Management*. ACM, New York, NY, USA, 52–63.
- SINGH, A. AND KICZALES, G. 2007. The scalability of AspectJ. In *Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research*. ACM, New York, NY, USA, 203–214.
- SMITH, A. J. 1982. Cache memories. *ACM Computing Surveys* 14, 3, 473–530.
- SOARES, S., LAUREANO, E., AND BORBA, P. 2002. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, New York, NY, USA, 174–190.
- Standard Performance Evaluation Corporation 2009. *SPECjbb2000: A Java Business Benchmark*. Standard Performance Evaluation Corporation. <http://www.spec.org/jbb2000/docs/whitepaper.html>.

- STEELE, G. L. 1975. Multiprocessing compactifying garbage collection. *Communications of the ACM* 18, 9, 495–508.
- STEIMANN, F. 2006. The paradoxical success of aspect-oriented programming. *SIGPLAN Not.* 41, 10, 481–497.
- STERRITT, R. AND HINCHEY, M. 2005. Autonomic computing - panacea or poppycock? In *Proceedings of the 12th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE Computer Society, Washington, DC, USA, 535–539.
- STONEBRAKER, M. 1981. Operating system support for database management. *Communications of the ACM* 24, 7, 412–418.
- SUN MICROSYSTEM, INC. 2000. J2ME building blocks for mobile devices. Tech. rep., Sun Microsystem, Inc.
- Sun Microsystems, Inc. 2006a. *JDKTM 5 Documentation*. Sun Microsystems, Inc. <http://java.sun.com/j2se/1.5/docs/>.
- Sun Microsystems, Inc. 2006b. *JDKTM 6 Documentation*. Sun Microsystems, Inc. <http://java.sun.com/javase/6/docs/>.
- Sun Microsystems, Inc. 2009a. *Enterprise JavaBeans Technology*. Sun Microsystems, Inc. <http://java.sun.com/products/ejb/>.
- Sun Microsystems, Inc. 2009b. *Java Data Objects (JDO)*. Sun Microsystems, Inc. <http://java.sun.com/jdo/index.jsp>.
- TANG, Y., GAO, Q., AND QIN, F. 2008. Leaksurvivor: towards safely tolerating memory leaks for garbage-collected languages. In *USENIX 2008 Annual*

- Technical Conference on Annual Technical Conference*. USENIX Association, Berkeley, CA, USA, 307–320.
- TANTER, E., SÉGURA-DEVILLECHAISE, M., NOYÉ, J., AND PIQUER, J. M. 2002. Altering Java semantics via bytecode manipulation. In *GPCE '02: Proceedings of the 1st ACM SIGPLAN/SIGSOFT conference on Generative Programming and Component Engineering*. Springer-Verlag, London, UK, 283–298.
- TATSUBORI, M. 2001. Separation of distribution concerns in distributed Java programming. In *Addendum to the 2001 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, Tampa, FL, USA.
- TIP, F., LAFFRA, C., SWEENEY, P. F., AND STREETER, D. 1999. Practical experience with an application extractor for Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 292–305.
- TUEL, W. G. 1976. An analysis of buffer paging in virtual storage systems. *IBM Journal of Research and Development* 20, 5 (Sep), 518–520.
- UNGAR, D. M. 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices* 19, 5 (Apr), 157–167.
- VAIDYANATHAN, K., HARPER, R. E., HUNTER, S. W., AND TRIVEDI, K. S. 2001. Analysis and implementation of software rejuvenation in cluster systems. *SIGMETRICS Performance Evaluation and Review* 29, 1, 62–71.
- VALLÉE-RAI, R., CO, P., GAGNON, E., HENDREN, L., LAM, P., AND SUN-DARESAN, V. 1999. Soot - a Java bytecode optimization framework. In *CAS-*

- CON '99: *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 13.
- VALLÉE-RAI, R., GAGNON, E., HENDREN, L. J., LAM, P., POMINVILLE, P., AND SUNDARESAN, V. 2000. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proceedings of the 9th International Conference on Compiler Construction*. Springer-Verlag, London, UK, 18–34.
- VENSTERMANS, K., EECKHOUT, L., AND BOSSCHERE, K. D. 2007. Java object header elimination for reduced memory consumption in 64-bit virtual machines. *ACM Transactions on Architecture and Code Optimization* 4, 3, 17.
- VENSTERMANS, K., EECKHOUT, L., AND DE BOSSCHERE, K. 2006. 64-bit versus 32-bit virtual machines for Java: Research articles. *Software Practice and Experience* 36, 1, 1–26.
- VILLAZÓN, A., BINDER, W., AND MORET, P. 2008. Aspect weaving in standard java class libraries. In *Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*. ACM, New York, NY, USA, 159–167.
- VITTER, J. S. 1998. External memory algorithms. In *PODS '98: Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. ACM, New York, NY, USA, 119–128.
- VITTER, J. S. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys* 33, 2, 209–271.
- W3C. 1998. The Document Object Model (DOM) Level 1 Specification: Version 1.0. Tech. Rep. REC-DOM-Level-1-19981001, World Wide Web Consortium (W3C). October.

- W3C. 2007. The XML Path Language (XPath) 2.0. Tech. Rep. REC-xpath20-20070123, World Wide Web Consortium (W3C). January.
- WANG, F., LI, J., AND HOMAYOUNFAR, H. 2007. A space efficient xml dom parser. *Data Knowl. Eng.* 60, 1, 185–207.
- WHITE, S. J. AND DEWITT, D. J. 1995. Quickstore: a high performance mapped object store. *The VLDB Journal* 4, 4, 629–673.
- WILSON, P. R. 1991. Pointer swizzling at page fault time: efficiently supporting huge address spaces on standard hardware. *SIGARCH Comput. Archit. News* 19, 4, 6–13.
- WILSON, P. R. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*. Springer-Verlag, London, UK, 1–42.
- WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1991. Effective “static-graph” reorganization to improve locality in garbage-collected systems. *ACM SIGPLAN Notices* 26, 6, 177–191.
- WILSON, P. R., LAM, M. S., AND MOHER, T. G. 1992. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*. ACM Press, San Francisco, CA, USA, 32–42.
- XU, G. AND ROUNTEV, A. 2008. Precise memory leak detection for Java software using container profiling. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, New York, NY, USA, 151–160.
- YANG, T., BERGER, E. D., KAPLAN, S. F., AND MOSS, J. E. B. 2006.

- CRAMM: virtual memory support for garbage-collected applications. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. USENIX Association, Berkeley, CA, USA, 103–116.
- YAO, B. B., ÖZSU, M. T., AND KEENLEYSIDE, J. 2003. Xbench - a family of benchmarks for xml dbmss. In *Proceedings of the VLDB 2002 Workshop EEXTT and CAiSE 2002 Workshop DTWeb on Efficiency and Effectiveness of XML Tools and Techniques and Data Integration over the Web-Revised Papers*. Springer-Verlag, London, UK, 162–164.
- YU, Z. C. H., LAU, F. C. M., AND WANG, C.-L. 2008. Object co-location and memory reuse for Java programs. *ACM Transactions on Architecture and Code Optimization* 4, 4, 1–36.
- ZAND, M., COLLINS, V., AND CAVINESS, D. 1995. A survey of current object-oriented databases. *SIGMIS Database* 26, 1, 14–29.
- ZHANG, L. AND KRINTZ, C. 2005. The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices. *ACM Transactions on Architecture and Code Optimization* 2, 2, 131–164.
- ZHANG, Y., ABU-KHZAM, F. N., BALDWIN, N. E., CHESLER, E. J., LANGSTON, M. A., AND SAMATOVA, N. F. 2005. Genome-scale computational approaches to memory-intensive applications in systems biology. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, 12.
- ZIGMAN, J. N. AND BLACKBURN, S. 1999. Java finalize method, orthogonal persistence and transactions. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8) and Proceedings of the 3rd International*

- Workshop on Persistence and Java (P JW3)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 363–369.
- ZILIO, D., LIGHTSTONE, S., LYONS, K., AND LOHMAN, G. 2001. Self-managing technology in IBM DB2 universal database. In *Proceedings of the Tenth International Conference on Information and Knowledge Management*. ACM, New York, NY, USA, 541–543.
- ZYL, P. V., KOURIE, D. G., AND BOAKE, A. 2006. Comparing the performance of object databases and ORM tools. In *Proceedings of the 2006 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries*. South African Institute for Computer Scientists and Information Technologists, Somerset West, Republic of South Africa, 1–11.