

# # 英語文献講読の資料まとめリポジトリ

英語文献講読に使った資料や当日の解説に使ったノートを保管する場所

## ## 2018-06-14

<dl>

<dt>タイトル</dt>

<dd>"Predictions of Solar Cycle 24 (Solar Physics2008, William Dean Pesnell)"</dd>

<dt>解説担当</dt>

<dd>柏田</dd>

<dt>参加者</dt>

<dd>柏田、才田、中村</dd>

<dt>講読箇所</dt>

<dd>Abstract (65 Words, 372 Characters)およびIntroduction (342 Words, 1842 Characters)</dd>

講読箇所を約20分かけて通読後、解説担当はこの文献から該当分野の課題、当該研究による課題解決手法、研究成果、を読み取って報告した。  
</dl>

## ## 2018-07-05

<dl>

<dt>タイトル</dt>

<dd>"The three-dimensional solar wind around solar maximum (GRL2003, McComas et. al.)"</dd>

<dt>解説担当</dt>

<dd>柏田</dd>

<dt>参加者</dt>

<dd>柏田、才田、中村</dd>

<dt>講読箇所</dt>

<dd>Abstract (149 Words, 770 Characters)およびIntroduction (801 Words, 4205 Characters)</dd>

文献を約40分かけて通読後、解説担当はこの文献から該当分野の課題、当該研究による課題解決手法、研究成果、を読み取って報告した。  
</dl>

## ## 2018-07-19

<dl>

<dt>タイトル</dt>

<dd>"Garbage-First Garbage Collection (ACM2004, Detlefs et. al.)"</dd>

<dt>解説担当</dt>

<dd>中村</dd>

<dt>参加者</dt>

<dd>才田、柏田、中村</dd>

<dt>講読箇所</dt>

<dd>Abstract (75 Words, 409 Characters)およびIntroduction (599 Words, 3390 Characters)</dd>

文献を約30分かけて通読後、解説担当はこの文献から該当分野の課題、当該研究による課題解決手法、研究成果、を読み取って報告した。  
</dl>

## ## 2018-10-10

<dl>

<dt>タイトル</dt>

<dd>"Interactive visualization of multi-dimensional data in R using OpenGL (Daniel Adler, 2002)"</dd>

<dt>解説担当</dt>

<dd>柏田</dd>

<dt>参加者</dt>

<dd>才田、柏田、中村</dd>

<dt>講読箇所</dt>

<dd>Introduction (580 Words, 3390 Characters)</dd>

文献を約30分かけて通読後、参加者全体で話し合い、この文献から該当分野の課題、当該研究による課題解決手法、研究成果、を読み取り考察した。  
</dl>

# Predictions of Solar Cycle 24

William Dean Pesnell

Received: 31 March 2008 / Accepted: 15 July 2008 / Published online: 21 August 2008  
© The Author(s) 2008. This article is published with open access at Springerlink.com

**Abstract** A summary and analysis of more than 50 predictions of the amplitude of the upcoming Solar Cycle 24 is presented. All of the predictions were published before solar minimum and represent our efforts to anticipate solar maximum at ever-earlier epochs. The consistency of the predictions within their assigned categories is discussed. Estimates of the significance of the predictions, compared to the climatological average, are presented.

## 1. Introduction

Solar-cycle predictions test our knowledge of the solar dynamo, a term that includes the processes involved in the production, transport, and destruction of solar magnetic field. Models of the dynamo are validated by their ability to predict solar activity over short and long timescales. Predictions of the magnitude and timing of Solar Cycle 24 are also used by a variety of space-weather groups to estimate orbital drag and other consequences of space weather in the upcoming cycle. Solar-activity predictions are used by space-weather operators to plan when to reboost satellites in low-Earth orbit, anticipate radiation exposure for current and upcoming missions, and to plan for outages in radio-based communication and navigation systems. Space-weather operators want to know the significance of each prediction when compared to other predictions.

Sunspot number ( $R_z$ ) is the most commonly predicted solar activity index. The rate of solar flares and amount of energy they release are well correlated with the sunspot number, as is the rate of coronal mass ejections. Cosmic rays, whose flux is anticorrelated with the solar cycle, are a significant source of radiation hazard in space. Geomagnetic activity has one component that is proportional to  $R_z$  and another, which can be a source of significant space weather, that resembles the sunspot number but shifted forward several years (about a quarter cycle). But, in general, the sunspot number (or a proxy index such as F10.7, the spectral irradiance at a radio wavelength of 10.7 cm) is the basic quantity needed for space-weather work. The amplitude of the annual-averaged sunspot number for Solar Cycle  $n$  will be called  $R_n$ .

---

W.D. Pesnell (✉)  
Code 671, NASA Goddard Space Flight Center, Greenbelt, MD, USA  
e-mail: [william.d.pesnell@nasa.gov](mailto:william.d.pesnell@nasa.gov)

We report here a summary and discussion of the predictions of Solar Cycle 24. The predictions are analyzed within categories to determine trends and consistencies. We also calculate whether each prediction would be considered significantly different from one of the simplest predictions: That Solar Cycle 24 will be a cycle of average amplitude and duration.

## 2. Predictions of Cycle 24

The Solar Cycle 24 Prediction Panel was convened in October 2006 to develop a consensus prediction of Solar Cycle 24. To ensure that a wide range of predictions were considered, predictions of the amplitude and timing of Solar Cycle 24 were solicited from the community in the categories of Fun, Precursor, Spectral, Climatology, Recent climatology, Neural network, Physics-based, or Other. The call for predictions was published in several newsletters, including the 15 August 2006 “Solar News” (<http://solarnews.nso.edu>).

The more than 50 predictions in Table 1 are a combination of 15 predictions submitted in response to the call by the Solar Cycle 24 Prediction Panel, four predictions described here, six predictions produced by the Prediction Panel, with the remaining predictions culled from the refereed literature. The table is organized by the predicted sunspot maximum and includes the predicted maximum sunspot number (value and timing), category of prediction, a short summary of method, and the reference. The third column of Table 1 contains a one-letter abbreviation of the category for each prediction. If a prediction was found during the literature search, the category was assigned by the author. No predictions were received or discovered in the categories Fun or Other. Another summary list of predictions was given by Janssens (2005, 2006). The category “Physics-based models” was renamed “Dynamo model” to more accurately reflect the nature of the expected predictions. Four predictions (Thompson (1993), aa\_min, aa\_4yr, and modified Feynman) were developed during the deliberations of the panel and two consensus predictions were released by the panel (Biesecker, 2007). The consensus predictions were not placed into categories. The references, predicted maxima, and uncertainties are shown in the bottom part of Figure 1. Categories for each prediction are shown by a color coding listed in the upper panel.

Some predictions of Solar Cycle 24 are not included in Table 1. A prediction by Li, Gao, and Su (2005) was omitted as they included multiple predictions, depending on the timing of solar minimum and rise time of Cycle 24. Some of their conditions have already passed; their remaining predictions are that the current solar minimum will be reached in June 2008 ( $\pm$  two months), the next maximum will be in February 2013 ( $\pm$  eight months), and the maximum will be about 137 or 80, depending on whether the cycle is a fast riser or a slow riser. Volobuev and Makarenko (2008) used a combination of the sunspot number and radiocarbon variations to estimate that the upcoming decade will be smaller than the last but did not convert this into a prediction of  $R_{24}$ .

It is necessary to have quantitative estimates of the uncertainty of the predictions – in both magnitude and timing. Most of the amplitude predictions included either an error estimate or a range that could be converted into an error estimate. Timing predictions were less precise and usually depend on the timing of solar minimum. As we move into Cycle 24, those timing predictions that depend on the time of minimum and the shape of the rise will become more accurate. Although the timing predictions are listed in Table 1, they will not be discussed further.

# Predictions of Solar Cycle 24

## Abstract

A summary and analysis of more than 50 predictions of the amplitude of the upcoming Solar Cycle 24 is presented.

来たる第 24 太陽周期の振幅の 50 以上の予測の要約および分析を示す。

All of the predictions were published before solar minimum and represent our efforts to anticipate solar maximum at ever-earlier epochs.

すべての予測は、太陽極小期の前に公表されており、これ以前の時期に太陽極大期を予測する努力を表明します。

The consistency of the predictions within their assigned categories is discussed.

割り当てられたカテゴリ内の予測の一貫性について説明します。

Estimates of the significance of the predictions, compared to the climatological average, are presented.

気候学的平均と比較して、予測の重要性の見積もりが提示される。

## 1. Introduction

Solar-cycle predictions test our knowledge of the solar dynamo, a term that includes the processes involved in the production, transport, and destruction of solar magnetic field.

太陽周期の予測は、太陽磁場の生成、輸送、破壊に関わるプロセスを含む期間の、太陽ダイナモの知識をテストします。

Models of the dynamo are validated by their ability to predict solar activity over short and long timescales.

ダイナモのモデルは、短期および長期の時間スケールにわたる太陽活動を予測する能力によって検証されます。

Predictions of the magnitude and timing of Solar Cycle 24 are also used by a variety of space-weather groups to estimate orbital drag and other consequences of space weather in the upcoming cycle.

第 24 太陽周期の大きさとタイミングの予測は、今後のサイクルにおける軌道抵抗(?)や宇宙天気のための帰結を推定するために、様々な宇宙天気グループ(?)によって使用されています。

Solar-activity predictions are used by space-weather operators to plan when to reboost satellites in low-Earth orbit, anticipate radiation exposure for current and upcoming missions, and to plan for outages in radio-based communication and navigation systems.

太陽活動の予測は、宇宙天気オペレータが、低軌道の衛星をいつ再起動するかを計画したり、現在および将来の任務の放射線被曝を予測したり、ラジオベースの通信システムやナビゲーションシステムの停止を計画するために使用されます。

### 【課題】

Space-weather operators want to know the significance of each prediction when compared to other predictions. 宇宙天気オペレータは、他の予測と比較した場合の、各予報の重要性を知りたいと考えています。

Sunspot number ( $R_z$ ) is the most commonly predicted solar activity index.

太陽黒点番号 ( $R_z$ ) は最も一般的に予測される太陽活動指標です。

The rate of solar flares and amount of energy they release are well correlated with the sunspot number, as is the rate of coronal mass ejections.

太陽フレアの速度およびそれらが放出するエネルギーの量は、コロナ質量放出の速度と同様に、太陽黒点の数とよく相関している。

Cosmic rays, whose flux is anticorrelated with the solar cycle, are a significant source of radiation hazard in space. (フラックスが太陽周期と相関性がない)

宇宙線は、宇宙における放射線被害の重大な原因である。

Geomagnetic activity has one component that is proportional to  $R_z$  and another, which can be a source of significant space weather, that resembles the sunspot number but shifted forward several years (about a quarter cycle).

地磁気活動は、 $R_z$  に比例する成分と、

重要な宇宙天気の原因となる別の成分があり、太陽黒点の数に似ているが、数年前(約 4 分の 1 周期)にシフトした成分を有する。

But, in general, the sunspot number (or a proxy index such as F10.7, the spectral irradiance at a radio wavelength of 10.7 cm) is the basic quantity needed for space- weather work.

しかし、一般に、太陽黒点の数(または 10.7cm の電波波長における分光放射照度である F10.7 のようなプロキシインデックス)は、宇宙天気の仕事に必要な基本量です。

The amplitude of the annual-averaged sunspot number for Solar Cycle  $n$  will be called  $R_n$ .

第  $n$  太陽周期の年間平均太陽黒点数の振幅を  $R_n$  と呼びます。

We report here a summary and discussion of the predictions of Solar Cycle 24.

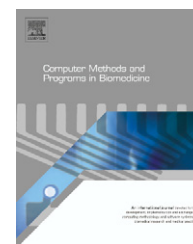
ここでは、サイクル 24 の予測の要約と考察を報告します。

The predictions are analyzed within categories to determine trends and consistencies.

予測は、カテゴリ内で分析され、傾向と一貫性を明らかにします。

We also calculate whether each prediction would be considered significantly different from one of the simplest predictions: That Solar Cycle 24 will be a cycle of average amplitude and duration.

また、各予測が最も単純な予測（太陽サイクル 24 が平均振幅と持続時間のサイクルになる）の 1 つと大きく異なると考えられるかどうかとも予想します。



# Web-based interactive 2D/3D medical image processing and visualization software

Seyyed Ehsan Mahmoudi<sup>a</sup>, Alireza Akhondi-Asl<sup>a,b</sup>, Roohollah Rahmani<sup>a,d</sup>,  
Shahrooz Faghih-Roohi<sup>a</sup>, Vahid Taimouri<sup>a</sup>, Ahmad Sabouri<sup>a</sup>,  
Hamid Soltanian-Zadeh<sup>a,b,c,\*</sup>

<sup>a</sup> Control and Intelligent Processing Center of Excellence (CIPCE), School of Electrical and Computer Engineering, University of Tehran, Tehran, Iran

<sup>b</sup> School of Cognitive Sciences, Institute for Research in Fundamental Sciences, Tehran, Iran

<sup>c</sup> Image Analysis Laboratory, Department of Radiology, Henry Ford Hospital, Detroit, MI, USA

<sup>d</sup> Microsoft Corporation, Redmond, WA, USA

## ARTICLE INFO

### Article history:

Received 6 May 2009

Received in revised form

16 November 2009

Accepted 19 November 2009

### Keywords:

Web-based software tools

2D and 3D processing and  
visualization

Medical imaging and analysis

## ABSTRACT

There are many medical image processing software tools available for research and diagnosis purposes. However, most of these tools are available only as local applications. This limits the accessibility of the software to a specific machine, and thus the data and processing power of that application are not available to other workstations. Further, there are operating system and processing power limitations which prevent such applications from running on every type of workstation. By developing web-based tools, it is possible for users to access the medical image processing functionalities wherever the internet is available. In this paper, we introduce a pure web-based, interactive, extendable, 2D and 3D medical image processing and visualization application that requires no client installation. Our software uses a four-layered design consisting of an algorithm layer, web-user-interface layer, server communication layer, and wrapper layer. To compete with extendibility of the current local medical image processing software, each layer is highly independent of other layers. A wide range of medical image preprocessing, registration, and segmentation methods are implemented using open source libraries. Desktop-like user interaction is provided by using AJAX technology in the web-user-interface. For the visualization functionality of the software, the VRML standard is used to provide 3D features over the web. Integration of these technologies has allowed implementation of our purely web-based software with high functionality without requiring powerful computational resources in the client side. The user-interface is designed such that the users can select appropriate parameters for practical research and clinical studies.

© 2009 Elsevier Ireland Ltd. All rights reserved.

\* Corresponding author at: Radiology Image Analysis Lab., Henry Ford Hospital, One Ford Place, 2F, Detroit, MI 48202, USA.  
Tel.: +1 313 874 4482; fax: +1 313 874 4494.

E-mail address: [hamids@rad.hfh.edu](mailto:hamids@rad.hfh.edu) (H. Soltanian-Zadeh).

0169-2607/\$ – see front matter © 2009 Elsevier Ireland Ltd. All rights reserved.

doi:10.1016/j.cmpb.2009.11.012



## 1. Introduction

Today medical imaging plays an important role in speed and quality of medical diagnosis. Conventional radiology is prone to problems such as losing films, accessing with delay, spending considerable amount of time and cost to copy and archive the film, and limited application of image processing methods [1]. Towards the end of the 1970s, digital methods for radiology which use digital image representation became widespread [2]. In digital radiology, images are either acquired digitally or converted from analog to digital. In either case, this digital input necessitates strong medical image processing tools to process them.

There are several common functionalities for any medical image processing system. Digital images should be processed, saved, and retrieved easily and quickly using the software. They must compromise their characteristics in terms of reading, writing, and representing different image formats, applying various automatic analysis methods on the images in 2D and 3D, and applying new image processing methods to accurately segment and visualize the data [3,4]. These functionalities are necessary for computer assisted diagnosis and therapy.

Various software for representing, processing, and visualization of medical images have been designed [5–30] including 3D-Doctor, eFilm Workstation, PACSPlus Viewer, EigenTool, MEDAL, Imaris, Caret, Analyze, Vitrea2-Fusion7D, Medx, 3DVIEWNIX, 3D Slicer, Julius, OsiriX, BrainSuite, MIPAV, and MRicro. Some of these software tools work as image viewers supporting a variety of image formats such as Dicom and Analyze [6,7]. Other tools also provide processing and visualization functionalities such as noise suppression, registration, applying conventional segmentation methods, analyzing images for diagnosis purposes, and representing 2D and 3D data using regular visualization methods. It should be noted that not all of these features are implemented in all of the software tools. Usually, each tool is developed such that it is suitable for some specific applications. For example, 3D-Doctor is developed for processing and visualization, while Vitrea-Fusion7D is developed for image registration.

Another important aspect of medical image processing software is the extendibility of the software. As new processing algorithms are developed which are more robust, powerful, and suitable for specific applications, extensions of the software to include these algorithms are inevitable. Among these existing tools, most are not extendible. 3D Slicer is unique as it is open source and the user may add new processing routines to it. This feature has made this software more practical and appropriate for academic and research applications.

An important issue in medical imaging is the ability to access the processed data quickly and easily. Web-based software tools are becoming a popular solution [22–30]. One such example is the Web-Based Multi-layer Visualization System (WBMVS) [30]. However, these web applications either suffer from poor user-interface (UI) functionalities or require the user to install ActiveX or Java Applet components. This causes the loss of many advantages of web-based applica-

tions. Using a web application, by centralized storage of data and availability of the software on different platforms, physicians can easily access patient's data from any computer station, regardless of relative processing power or operating system.

There are some fundamental advantages in using web-based medical software. Because of rapid changes in medical tools, the ability to update the software without interrupting the users and modifying their hardware and local software is necessary. Further, there are many complex processing algorithms that require large memory and computational resources that may not be available on many client machines [30]. By centralizing processing power and using server machines with large computational resources, it is possible for the client machine to work as a simple terminal. The acquired data can be saved on the server side and users can access them via the network. The importance of this structure is particularly evident in emergency conditions where quick decisions should be made in locations where resources are limited, such as outside of the hospital or in the field. Physicians can access and process the patient's medical data quickly and easily using the proposed web-based medical imaging software.

Although there are many Picture Archiving and Communication Systems (PACS) available for centralized data access, most of these systems are designed for Local Area Network (LAN) access, not for internet use. Also, all current web-based PACS systems are based on ActiveX or Java Applet, which are not considered pure web-based solutions, because they impose pre-installation requisites and are not platform independent. In addition, current PACS solutions normally do not perform algorithmic processing on the image and simply serve as image viewers.

In this paper, we introduce a pure web-based, interactive, extendable, 3D and 2D medical image processing and visualization application, which requires no client installation. Our software uses a four-layered design consisting of an algorithm layer, web-UI layer, server communication layer, and wrapper layer. To compete with extendibility of the current local medical image processing software, each layer is highly independent of the other layers. A wide range of medical image preprocessing, registration, and segmentation methods are implemented using open source libraries. Desktop-like user interaction is provided by using AJAX technology in the Web-UI. For the visualization functionality of the software, the VRML standard is used to provide 3D features over the web. Integration of these technologies has allowed implementation of our purely web-based software with high functionality without requiring powerful computational resources in the client side. The UI is designed such that the users can select appropriate parameters for practical research and clinical studies.

The organization of the rest of the paper is as follows. In Section 2, we describe design constraints and technology choices. Section 3 describes our software architecture. Section 4 describes the user-interface of the software. Section 5 compares our software with software tools developed previously for representing, processing, and visualization of medical images. Finally, in the Section 6, we present discussion and conclusions.

# Interactive visualization of multi-dimensional data in R using OpenGL

6-Monats-Arbeit im Rahmen der Prüfung für  
Diplom-Wirtschaftsinformatiker an der Universität  
Göttingen

vorgelegt am 09.10.2002  
von Daniel Adler  
aus Göttingen

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Approach . . . . .	2
1.2	Further reading . . . . .	2
<b>2</b>	<b>R</b>	<b>4</b>
2.1	Language . . . . .	4
2.2	Graphical subsystem . . . . .	5
2.2.1	Graphics plots . . . . .	6
2.2.2	Devices . . . . .	6
2.3	Documentation . . . . .	7
2.4	Extensions . . . . .	8
2.5	Distribution . . . . .	8
<b>3</b>	<b>Interactive visualization</b>	<b>9</b>
3.1	Geometry-based Computer graphics . . . . .	9
3.2	Homogeneous coordinates and transformations . . . . .	10
3.3	Lighting . . . . .	11
3.3.1	Shading . . . . .	11
3.3.2	Material . . . . .	12
3.4	Real-time Rendering . . . . .	12
3.4.1	Application stage . . . . .	13

3.4.2	Geometry stage . . . . .	13
3.4.3	Rasterization stage . . . . .	15
3.5	OpenGL . . . . .	17
3.5.1	State machine . . . . .	17
3.5.2	Client/Server architecture . . . . .	17
3.5.3	Display lists . . . . .	18
3.5.4	Vertex arrays . . . . .	18
3.5.5	Windowing system interface . . . . .	18
3.6	Interaction . . . . .	18
<b>4</b>	<b>Functional Design</b>	<b>19</b>
4.1	Goals . . . . .	19
4.2	Device . . . . .	20
4.3	Model . . . . .	20
4.3.1	Coordinate system . . . . .	22
4.3.2	Scene database . . . . .	23
4.3.3	Shape . . . . .	24
4.3.4	Viewpoint . . . . .	25
4.3.5	Light . . . . .	26
4.3.6	Background . . . . .	26
4.3.7	Bounding box . . . . .	27
4.3.8	Appearance . . . . .	28
4.4	API . . . . .	30
4.4.1	Device management functions . . . . .	31
4.4.2	Scene management functions . . . . .	32
4.4.3	Export functions . . . . .	32
4.4.4	Shape functions . . . . .	32
4.4.5	Environment functions . . . . .	33

4.4.6	Appearance . . . . .	34
<b>5</b>	<b>Software Development</b>	<b>36</b>
5.1	Object-orientation . . . . .	36
5.1.1	Notation using the UML . . . . .	37
5.1.2	Software patterns . . . . .	38
5.1.3	C++ . . . . .	39
5.2	Analysis . . . . .	40
5.2.1	Shared library . . . . .	41
5.2.2	Windowing system . . . . .	42
5.3	Architecture . . . . .	44
5.4	Foundation layer . . . . .	44
5.4.1	<i>types</i> module . . . . .	45
5.4.2	<i>math</i> module . . . . .	46
5.4.3	<i>pixmap</i> module . . . . .	52
5.4.4	<i>gui</i> module . . . . .	52
5.4.5	<i>lib</i> module . . . . .	54
5.5	<i>scene</i> module . . . . .	55
5.5.1	Database . . . . .	55
5.5.2	Rendering . . . . .	58
5.6	Device . . . . .	66
5.6.1	<i>rglview</i> module . . . . .	66
5.6.2	<i>device</i> module . . . . .	67
5.7	Client . . . . .	68
5.7.1	<i>api</i> module . . . . .	68
5.7.2	<i>devicemanager</i> module . . . . .	68
5.8	API implementation . . . . .	69
5.8.1	Interface and data passing between R and C . . . . .	70

5.8.2	R functions . . . . .	70
<b>6</b>	<b>Examples</b>	<b>72</b>
6.1	Launching RGL . . . . .	72
6.2	Statistical data analysis . . . . .	73
6.2.1	Estimating animal abundance . . . . .	73
6.2.2	Kernel smoothing . . . . .	73
6.2.3	Real-time animations . . . . .	74
6.2.4	Image generation for animation . . . . .	74
<b>7</b>	<b>Summary and Outlook</b>	<b>75</b>
7.1	Summary . . . . .	75
7.2	Outlook . . . . .	76
7.2.1	Ports . . . . .	76
7.2.2	Rendering improvements . . . . .	76
7.2.3	Functionality improvements . . . . .	77
7.2.4	Scene improvements . . . . .	77
7.2.5	GUI improvements . . . . .	77
7.2.6	R programing interface improvements . . . . .	78

# List of Figures

4.1	Scene objects . . . . .	21
4.2	Coordinate system . . . . .	22
4.3	Face orientation . . . . .	22
4.4	Logical database model . . . . .	23
4.5	Shape objects . . . . .	24
4.6	<i>Bounding Box</i> tick-marks . . . . .	28
4.7	Overview of the RGL API . . . . .	31
5.1	UML notations . . . . .	38
5.2	Device Input/Output . . . . .	41
5.3	Overview of C++ modules . . . . .	44
5.4	Class diagram: Double-linked lists . . . . .	45
5.5	Geometry transformation of polar coordinates . . . . .	48
5.6	Perspective viewing volume frustum . . . . .	49
5.7	Top-view of frustum enclosing a bounding sphere volume . . . . .	50
5.8	<i>gui</i> classes . . . . .	53
5.9	<i>printMessage()</i> on Win32 platform . . . . .	54
5.10	Class hierarchy of the scene database . . . . .	56
5.11	Bounding box mesh structure (left) and example with contours and axis (right) . . . . .	64
7.1	Graphical user-interface component <i>FourView</i> . . . . .	77

# List of Tables

3.1	Vertex attributes . . . . .	14
3.2	Light attributes . . . . .	14
4.1	<i>Viewpoint</i> parameters . . . . .	25
4.2	Dragging actions for interactive viewpoint navigation . . . . .	25
4.3	<i>Light</i> parameters . . . . .	26
4.4	<i>Background</i> parameters . . . . .	26
4.5	<i>Bounding Box</i> parameters . . . . .	27
4.6	Appearance parameters . . . . .	29
5.1	Data type mapping between R and C . . . . .	70



# Chapter 1

## Introduction

The visualization of empirical and simulated data in conjunction with function graphs is a common technique for scientists to identify correlations in multivariate data, to compare distributions with known distribution functions and to illustrate state facts.

2D visualizations such as scatterplots, distribution and density function plots, histograms and pair plots are useful graphical techniques for interactive data analysis.

3D visualizations provide an additional coordinate axis. Due to the fact that 3D visualizations are projected on 2D screen displays, *interactive viewpoint navigation* is required to get an adequate overview by exploring the 3D world. In contrast, 2D visualizations do not require special navigation facilities.

Three variable visualizations and the usage of modern graphics drawing techniques like transparency give scientists abilities at hand to analyse multivariate data.

The current R graphics capability lacks the ability to interactively visualize data in 3D and has limited support for 3D graphics plots. Interactive visualization requires a certain degree of graphics rendering speed which optimized engines should deliver.

This work documents the development of a R package named “RGL”. It contains an interactive visualization device system with an R programming interface. RGL uses the OpenGL library as the rendering backend providing an interface to graphics hardware. Most graphics hardware vendors pro-

vide an OpenGL driver for their hardware systems and even software-only implementations exist.

## 1.1 Approach

The R system is widely used throughout the statistical community. RGL extends R with a graphics device system that includes new graphical capabilities. By adapting some common programming interface designs from the current R graphics system, users need less time to study the usage of RGL.

This leads to the approach of analysing R graphics capabilities and its programming interface to derive requirements and guidelines for the software design.

The software concept evolves using a scene database oriented approach to design the functionality.

The development has been done using the methodology of object-oriented software development and an object-oriented programming language. The advantage is a solid transition from the software concept to design, and finally to implementation. The object-orientation provides techniques that structure software systems logically using modularization, abstraction, data encapsulation and polymorphism, which leads to a reduction of complexity. The methodology of *Software Patterns* has been applied to design a solid architecture.

## 1.2 Further reading

Chapter 2 gives an overview of the R environment and a brief introduction into key features of the R language. A short overview of the graphics capabilities and programming interface is presented. Furthermore, the extension mechanism of R is discussed.

Chapter 3 gives an introduction into the field of interactive visualization. Real-time rendering will be outlined together with an introduction to OpenGL.

Chapter 4 describes the functionality of the device, the graphics rendering model and explains the application programming interface using a top-down

approach.

In Chapter 5 the development process is presented. The chapter starts with an introduction to the object-oriented methodology including the C++ language, UML notations and *Software Patterns*. The R extension mechanism via shared libraries and a discussion about windowing systems give the outline for the architecture design. A detailed bottom-up description on a module- and class-level with details on method implementations round up the C++ implementation. The RGL API implementation in R completes the software system documentation.

Chapter 6 gives some examples to illustrate features of RGL package.

Chapter 7 gives a summary and a future outlook of further development plans.

# Chapter 2

## R

*R* is an open-source implementation of the *S* and its successor *Splus*. It is a statistical computation environment with an interpreted programming language, an interactive command line interface, a graphics plotting device system, a documentation system and a package mechanism. Add-on packages extend the system and a network supports the distribution. It has been ported to three major platforms (Microsoft Windows, Apple Macintosh and to the X11 windowing system running across a variety of operating systems, including Linux and BSD derivatives).

### 2.1 Language

The *R language* is a dialect of *S*, which was designed in the 1980s and has been in widespread use in the statistical community. This section focuses on some remarkable properties of the language. Detailed information about *S* is given in [3]. The R Language Reference is distributed with the R software and is available online. [18]

The R language is an interpreted language used with an interactive command line interface. R provides different data types to store data in memory.

*Vectors* represent variable number of data elements of a specific storage data type. Supported storage data types are *logical*, *integer*, *double*, *complex* and *character*.

*Lists* are generic vectors containing elements of varying data types. They

can be attributed with *names* used for named indexing.

The *function* data type is the elementary execution unit that extends the R language with new functionality. Functions are defined using an argument list for input data and a body of R statements implementing the functionality probably providing an object as return value. The arguments can have default values. Functions are called using the object name with arguments that are given either in order or can be given as *tagged arguments*. That is, the data to be passed to the function is labeled using the form *label=value*. When combining default values and using tagged arguments, the number of arguments and its position in the list can vary.

The “...” (dot-dot-dot) data type is a *List* with unknown elements. It is commonly used in function argument list definitions to dispatch arguments to different specialized functions. Functions can pass the “...” object to subfunctions where it gets actually evaluated. When providing a standardized list of arguments for a group of public functions, the “...” object can be used to dispatch the arguments to an internal function that actually evaluates the arguments. When changing the interface, only the internal function must be modified.

R supports object-orientation using generic functions and specializing functions for a special class. Generic functions dispatch the control flow to specialized functions depending on the class of an object.

A *recycling rule* is used by operators that requires its operands to be of the same length. In case that the operands differ in length, they are recycled to the length of the longest.

## 2.2 Graphical subsystem

The graphical subsystem consists of a device system, device drivers for different plotting devices and an API <sup>1</sup>.

---

<sup>1</sup>Application Programming Interface

### 2.2.1 Graphics plots

The R graphics plotting facility provides a variety of graphs for data analysis. They can be divided into low-level and high-level functions for plotting. A generic interface for setting graphical parameters is supported across the majority of commands using the `...` data type.

Many plotting functions are generic functions with a default function implementation which are suffixed with `'.default'`.

The `plot()` function is a generic plot function which produces different graphs depending on the class of the given data.

Multivariate data is plotted using `pairs()` providing two specialized pair plots. `qqnorm()`, `qqline()` and `qqplot()` are distribution-comparison plots. `hist()` produces histograms and provides two specialized histogram plots. `dotchart()` constructs a dotchart of the data. `image()`, `contour()` and `persp()` are used to plot multivariate data using three variables.

The low-level plotting commands give full control over the complete graphics tasks and can be used to implement new types of graphs or to extend the graph. Display limits on a per axis base must be defined at the beginning of a plot and are fixed throughout the session. If the graphics plot hits the display limits, it gets clipped.

Interaction commands can be used to identify (`identify()`) or locate (`locate()`) datasets in an existing plot using the pointing device of the windowing system.

`plot.new()` and `frame()` complete the current plot, if there is one and start a new plot.

The `par()` command is used to set or query graphical parameters. A group of parameters is defined as read-only and can not be modified. The other parameters can be set by `par()` explicitly or can be passed to `plot()`, `points()`, `lines()`, `axis()`, `title()`, `text()` and `mtext()`.

### 2.2.2 Devices

Graphics devices in the sense of R are graphics plotting devices. The R devices can be divided into interactive and batch mode devices. Interactive

devices immediately display plots, while batch mode devices output the plot to a file. A variety of device drivers are implemented. All supported windowing systems (currently Win32, Macintosh, X11) provide an interactive device driver that is capable of displaying the output using the graphics facilities of the windowing system. Batch mode devices are provided for the following output file formats: postscript, pdf, pictex, png, jpeg, bmp, xfig and bitmap. R devices can be configured to contain several plots layouted on the display. This is useful for comparing multiple plots.

A sample plot session would be carried out like this:

1. Open a device
2. Optionally setup device parameters (e.g. multiple plots)
3. Start a new plot by setting up axis limits
4. Post a sequence of graphics plotting instructions
5. Optionally, start a new plot
6. Close device

R has an interesting behaviour for interactive plots. In case of a new plotting command, it automatically opens an interactive device if none is already opened.

## 2.3 Documentation

R is delivered with comprehensive documentation about the language[18], writing extensions[19] and data import/export[17]. Functions, data sets and general features are documented in the *Rd documentation format*, which is part of the R distribution. The Rd format is a subset of  $\text{\LaTeX}$ , representing a meta-format for a variety of documentation formats. Transformation tools are provided that generate HTML,  $\text{\LaTeX}$ , pdf, Windows HTML Help Format and R manual pages. The R manual pages are viewed by a browser component of the R system. The Rd format provides the writing of examples when documenting functions. Additionally the `example(topic)` command can be issued, which will execute example code in the manual page indexed by *topic*.

## 2.4 Extensions

R extensions are delivered as *R packages*. A package must provide a description. Optionally, it can contain R functions, datasets, documentation, shared libraries and additional files. The description provides informations about version, authors, dependencies to other packages and copyright issues. The documentation is written in the Rd meta-documentation format. Shared libraries are used to extend R with foreign code. R is able to interface to C or Fortran code in a shared library. A package optionally contains the source code to build the libraries or pre-compiled binaries for a specific platform. A utility tool suite written in *Perl* helps automating the process of building, verification and installation of packages. The packaging standard helps implementators to deliver functionality, datasets and documentation in a portable way.

## 2.5 Distribution

Packages are distributed in the *Comprehensive R Archive Network* (CRAN) that is accessable through the internet. Package authors upload package releases. R provides an automated download and install mechanism to install packages from the CRAN.



# Interactive visualization of multi-dimensional data in R using OpenGL

## Chapter 1

### Introduction

The visualization of empirical and simulated data in conjunction with function graphs is a common technique for scientists to identify correlations in multivariate data, to compare distributions with known distribution functions and to illustrate state facts.

関数グラフと（組み合わせた）実験(経験)データやシミュレーションデータの可視化は多変数データの相関性を確認したり、分布を既知の分布関数と比較したり、状態事実を図解したりするための科学者の共通のテクニックです。

2D visualizations such as scatterplots, distribution and density function plots, histograms and pair plots are useful graphical techniques for interactive data analysis.

散布図、密度関数プロット、ヒストグラム、ペアプロットとしての2D可視化はインタラクティブなデータ解析のための役に立つグラフィカルテクニックです。

3D visualizations provide an additional coordinate axis.

Due to the fact that 3D visualizations are projected on 2D screen displays, interactive viewpoint navigation is required to get an adequate overview by exploring the 3D world.

In contrast, 2D visualizations do not require special navigation facilities.

3D可視化は追加の座標軸を提供する。

2Dスクリーンディスプレイに映し出される3D可視化の事実のために、インタラクティブな視点ナビゲーションは3D世界を探索することによって十分な概観を得るために要求される。

対照的に、2D可視化は特別なナビゲーション機能を必要としない。

Three variable visualizations and the usage of modern graphics drawing techniques like transparency give scientists abilities at hand to analyse multivariate data.

3変数の可視化や透かしのような現代的なグラフィック描画技術の使用は科学者に多変数データの解析を手近にする能力を与える。

The current R graphics capability lacks the ability to interactively visualize data in 3D and has limited support for 3D graphics plots.

現在のRのグラフィック能力は3Dデータをインタラクティブに可視化する能力を欠き、3Dグラフィックプロットのためのサポートは限定的です。

Interactive visualization requires certain degree of graphics rendering speed which optimized engines should deliver.

インタラクティブな可視化は最適化したエンジンが提供するべきある程度のグラフィックレン

ダリンスピードを必要とする。

This work documents the development of a R package named "RGL".  
ここでは“RGL”と名付けたRのパッケージの開発を記録する。

It contains an interactive visualization device system with an R programming interface.  
それはRプログラミングインターフェースと一緒にインタラクティブな可視化デバイスシステムを含みます。

RGL uses the OpenGL library as the rendering backend providing an interface to graphics hardware.  
RGLはグラフィックハードウェアにインターフェースを提供しているレンダリングバックエンドとしてオープンGLライブラリを利用します。

Most graphics hardware vendors provide an OpenGL driver for their hardware systems and even software-only implementations exist.  
ほとんどのグラフィックハードウェア販売者はそれらのハードウェアシステム用のオープンGLドライバーを提供し、ソフトウェアのみの実装さえ存在します。

*Mesa*

## 1. Approach

The R system is widely used throughout the statistical community.  
Rシステムは統計学コミュニティの至る所で広く利用されている。

RGL extends R with a graphics device system that includes new graphical capabilities.  
RGLは新たなグラフィカル能力を含むグラフィックデバイスシステムでRを拡張している。

By adapting some common programming interface designs from the current R graphics system, users need less time to study the usage of RGL.  
現在のRグラフィックシステムからいくつかの一般的なプログラミングインターフェースデザインを適合することによって、利用者はRGLの利用方法を学習する時間を減らすことができる。

This leads to the approach of analyzing R graphics capabilities and its programming interface to derive requirements and guidelines for the software design.  
これにより、Rグラフィックス性能とそのプログラミングインターフェースを分析したり、ソフトウェア設計の要件とガイドラインを導き出したりするアプローチにつながります。

The software concept evolves using a scene database oriented approach to design the functionality.  
ソフトウェアの概念は、機能を設計するためのシーンデータベース指向のアプローチを使用し  
て進化しています。

The development has been done using the methodology of object-oriented software development and an object-oriented programming language.  
開発はオブジェクト指向ソフトウェア開発の方法論とオブジェクト指向のプログラミング言語

を使用しています。

The advantage is a solid transition from the software concept to design, and finally to implementation. 利点は、ソフトウェアコンセプトから設計、そして最終的には実装への堅実な移行です。

The object-orientation provides techniques that structure software systems logically using modularization, abstraction, data encapsulation and polymorphism, which leads to a reduction of complexity.

オブジェクト指向は、モジュール化、抽象化、データカプセル化、および多形性を論理的に使用してソフトウェアシステムを構造化する技術を提供し、複雑さの低減につながる。

The methodology of Software Patterns has been applied to design a solid architecture.

ソフトウェアパターンの方法論は、強固なアーキテクチャを設計するために適用されています。

## 1.2 Further reading

Chapter 2 gives an overview of the R environment and a brief introduction into key features of the R language. A short overview of the graphics capabilities and programming interface is presented. Furthermore, the extension mechanism of R is discussed.

第2章では、R環境の概要と、R言語の主要機能の概要について説明します。グラフィックス機能とプログラミングインタフェースの概要を示します。さらに、Rの拡張機構について議論する。

Chapter 3 gives an introduction into the field of interactive visualization. Real-time rendering will be outlined together with an introduction to OpenGL.

第3章では、インタラクティブな可視化の分野について紹介します。リアルタイムレンダリングについては、OpenGLの紹介とともに概説します。

Chapter 4 describes the functionality of the device, the graphics rendering model and explains the application programming interface using a top-down approach.

第4章では、デバイスの機能、グラフィックスレンダリングモデル、およびトップダウンアプローチを使用したアプリケーションプログラミングインターフェイスについて説明します。

In Chapter 5 the development process is presented. The chapter starts with an introduction to the object-oriented methodology including the C++ language, UML notations and Software Patterns. The R extension mechanism via shared libraries and a discussion about windowing systems give the outline for the architecture design. A detailed bottom-up description on a module- and class-level with details on method implementations round up the C++ implementation. The RGL API implementation in R completes the software system documentation.

第5章では開発プロセスを紹介します。この章では、C++言語、UML表記、ソフトウェアパターンなどのオブジェクト指向メソッドロジについて説明します。共有ライブラリによるR拡張

メカニズムとウィンドウシステムに関する議論は、アーキテクチャ設計のアウトラインを示しています。メソッドの実装に関する詳細を含むモジュールおよびクラスレベルの詳細なボトムアップの説明は、C++実装をラウンドアップします。RのRGL APIの実装は、ソフトウェアシステムのドキュメントを完成させます。

Chapter 6 gives some examples to illustrate features of RGL package.

第6章では、RGLパッケージの機能を説明するいくつかの例を示します。

Chapter 7 gives a summary and a future outlook of further development plans.

第7章では、今後の開発計画の概要と今後の展望について述べる。

## Chapter 3

### Interactive visualization

Interactive visualization is common in the field of computer aided design, scientific visualization and computer games. A computer graphics system is the core providing computer-generated images.

インタラクティブな可視化は設計、科学の可視化、コンピュータゲームを助けるコンピュータの共通分野です。コンピュータグラフィックシステムはコンピュータが作り出す画像を提供する核となっています。

Angel[2] describes five major components that can be found in a computer graphics system: Processor, Memory, Frame buffer, Output device and Input devices. At present, almost all graphics systems are raster based. A picture is produced as an array – the raster – of picture elements, or pixels, within the graphics system.

エンジェルはコンピュータグラフィックシステム内で見つけられる五つの有名な要素（プロセッサ、メモリー、フレームバッファ、出力デバイス、入力デバイス）を述べる。現在、ほとんど全てのグラフィックシステムはラスターが元になっている。写真はグラフィックシステム内部で、画像素子もしくは画素の行列(ラスター)として提供される。

### 3.1 Geometry-based Computer graphics

Geometry-based computer graphics use three-dimensional primitives such as points, lines, triangles, quadrilaterals and polygons as basic building blocks to construct complex objects.

幾何学ベースのコンピュータグラフィックは複雑なオブジェクトを構築するための基礎構成要素として点、線、三角形、四角形、多角形のような3次元の原始関数を使います。

The primitives are described using vertices in a local coordinate space.

その原始関数は局所的な座標空間の頂点を利用して述べられている。

The object is transformed using translation, rotation and scaling to map it into the eye coordinate space, where it actually gets rendered.

オブジェクトは実際にレンダリングされる視線座標空間に対応させるために平行移動、回転、スケーリングを使った変換が行われる

Attributes such as color, material properties, normal vectors, edge flags and texture coordinates are associated with the vertex.

色、材料特性、法線ベクトル、エッジフラグ、テクスチャ座標などの属性は、頂点に関連付けられます。

This section focuses on geometry-based computer graphics used for high-performance rendering.

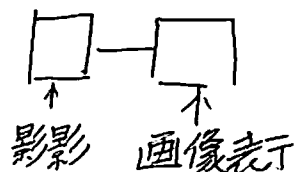
このセクションではハイパフォーマンスレンダリングのために使用される幾何学ベースのコンピュータグラフィックに焦点を当てる。

For detailed information on 3D computer graphics, including ray-tracing and ray-casting, see[21].

レイトレーシングとレイキャスティングを含む3Dコンピュータグラフィックの詳細情報は[21]をご覧ください。

# Web-based interactive 2D/3D medical image processing and visualization software

CT画像を表示するためのソフトウェア



## Abstract

There are many medical image processing software tools available for research and diagnosis purposes.

研究・診断目的で利用可能な医療画像処理ソフトウェアツールがたくさんある。

However, most of these tools are available only as local applications.

しかしながら、それらツールの大半がローカルアプリケーションとしてだけで利用可能だ。

This limits the accessibility of the software to a specific machine, and thus the data and processing power of that *local application* であること. application are not available to other workstations.

この特定の機械へのソフトウェアのアクセス制限により、データとアプリケーションの処理能力は他のワークステーションに利用できない。

Further, there are operating system and processing power limitations which prevent such applications from running on every type of workstation.

さらに、ワークステーションのあらゆる型で動作を妨げる、OS と処理能力のアプリケーションのような制限がある。

By developing web-based tools, it is possible for users to access the medical image processing functionalities wherever the internet is available.

Web ベースのツールの開発により、インターネットが利用可能な場所なら、ユーザーは医療画像を処理する機能にアクセス可能だ。

In this paper, we introduce a pure web-based, interactive, extendable, 2D and 3D medical image processing and visualization application that requires no client installation.

この論文では、クライアントのインストールを必要としない、純粋な Web ベースで、インタラクティブで、拡張可能で 2D や 3D な医療画像処理と可視化アプリケーションを紹介する。

Our software uses a four-layered design consisting of an algorithm layer, web-user-interface layer, server communication layer, and wrapper layer.

我々のソフトウェアはアルゴリズム層・Web ユーザーインターフェース層・サーバー通信層・ラッパー層から成る 4 層デザインを使用する。

To compete with extendibility of the current local medical image processing software, each layer is highly

independent of other layers.

現在のローカル医療画像処理ソフトウェアの拡張性と競合するため、それぞれの層は他の層と高く独立している。

A wide range of medical image preprocessing, registration, and segmentation methods are implemented using open source libraries.

広範囲の医療画像処理、登録、分割方式はオープンソースライブラリの実行される。

Desktop-like user interaction is provided by using AJAX technology in the web-user-interface.

デスクトップのようなユーザーとの対話処理は Web ユーザーインターフェースの AJAX 技術を利用して提供される。

For the visualization functionality of the software, the VRML standard is used to provide 3D features over the web. ソフトウェアの可視化機能のため、VRML スタンドアードは web 上で 3D 特徴を提供するために使用される。

↳ X3D → ?

Integration of these technologies has allowed implementation of our purely web-based software with high functionality without requiring powerful computational resources in the client side.

これら技術の統合はクライアント側の強力な計算資源の要求なしに、高い機能性で私たちの Web ベースのソフトウェアの実行を許している。

The user-interface is designed such that the users can select appropriate parameters for practical research and clinical studies.

ユーザーインターフェースは実用化研究や臨床研究のためユーザーが適切なパラメーターを選択できるようにデザインされた。

## Introduction

Today medical imaging plays an important role in speed and quality of medical diagnosis.

今日、医療画像は医療診断のスピードと質において重要な役割を担っている。

Conventional radiology is prone to problems such as losing films, accessing with delay, spending considerable amount of time and cost to copy and archive the film, and limited application of image processing methods [1].

従来の放射線医学では、フィルムの消失、遅れたアクセス、フィルムをコピーして保管するためのかなりの時間と費用の費やし、画像処理方法の限定された適用などの問題が生じやすい[1]。

Towards the end of the 1970s, digital methods for radiology which use digital image representation became

widespread [2].

1970 年代の終わりに向けて、デジタル画像表現を使用する放射線医学のためのデジタル手法が広く普及した[2]。

In digital radiology, images are either acquired digitally or converted from analog to digital.

デジタル放射線医学では、画像はデジタル方式で取得されるか、アナログからデジタルに変換される。

In either case, this digital input necessitates strong medical image processing tools to process them.

いずれの場合も、このデジタル入力、それら処理するために強力な医療画像処理ツールを必要とする。

There are several common functionalities for any medical image processing system.

どの医療画像処理システムにも、いくつかの共通の機能がある。

Digital images should be processed, saved, and retrieved easily and quickly using the software.

デジタル画像は、ソフトウェアを使用して簡単かつ速く処理、保存、検索される必要がある。

They must compromise their characteristics in terms of reading, writing, and representing different image formats, applying various automatic analysis methods on the images in 2D and 3D, and applying new image processing methods to accurately segment and visualize the data [3,4].

これらは、さまざまな画像形式の読み書き、表現、2D および 3D 画像のさまざまな自動分析方法の適用、データを正確にセグメント化して視覚化するための新しい画像処理方法を適用するという点で、特性を妥協する必要がある。

These functionalities are necessary for computer assisted diagnosis and therapy.

これらの機能は、コンピュータ支援による診断と治療に必要だ。

Various software for representing, processing, and visualization of medical images have been designed [5–30] including 3D-Doctor, eFilm Workstation, PACSPlus Viewer, EigenTool, MEDAL, Imaris, Caret, Analyze, Vitrea2-Fusion7D, Medx, 3DVIEWNIX, 3D Slicer, Julius, OsiriX, BrainSuite, MIPAV, and MRIcro.

3D-Doctor、eFilm Workstation、PACSPlus Viewer、EigenTool、MEDAL、Imaris、Caret、Analyze、Vitrea2-Fusion7D、Medx、3DVIEWNIX、3D Slicer、Julius、OsiriX、BrainSuite、MIPAV、MRIcro など医療画像の表示、処理、視覚化のためのさまざまなソフトウェアが設計されている。

Some of these software tools work as image viewers supporting a variety of image formats such as Dicom and Analyze [6,7].

これらのソフトウェアツールの一部は、Dicom や Analyze [6,7] のようなさまざまなイメージフォーマットをサポートするイメージビューアとして機能する。



Other tools also provide processing and visualization functionalities such as noise suppression, registration, applying conventional segmentation methods, analyzing images for diagnosis purposes, and representing 2D and 3D data using regular visualization methods.

その他のツールもまた、ノイズ抑制、登録、従来のセグメンテーション手法の適用、診断目的の画像の解析、および通常の視覚化手法を使用した 2D および 3D データの表現といった処理および視覚化機能を提供する。

It should be noted that not all of these features are implemented in all of the software tools.

これらの機能のすべてがすべてのソフトウェアツールで実装されているわけではない。

Usually, each tool is developed such that it is suitable for some specific applications.

通常、各ツールは特定のアプリケーションに適したものになるように開発されている。

For example, 3D-Doctor is developed for processing and visualization, while Vitera-Fusion7D is developed for image registration.

例えば、3D-Doctor は処理と視覚化のために開発されているのに対して、Vitera-Fusion7D は画像登録のために開発されている。

Another important aspect of medical image processing software is the extendibility of the software.

医療画像処理ソフトウェアのもう 1 つの重要な側面は、ソフトウェアの拡張性だ。

As new processing algorithms are developed which are more robust, powerful, and suitable for specific applications, extensions of the software to include these algorithms are inevitable.

より堅牢で強力で、特定のアプリケーションに適した新しい処理アルゴリズムが開発されるにつれて、これらのアルゴリズムを含むソフトウェアの拡張は避けられない。

Among these existing tools, most are not extendible.

これらの既存のツールの中でも、ほとんどが拡張可能ではない。

3D Slicer is unique as it is open source and the user may add new processing routines to it.

3D Slicer はオープンソースなのでユニークで、ユーザーは新しい処理ルーチンを追加することができる。

This feature has made this software more practical and appropriate for academic and research applications.

この特徴により、このソフトウェアは、学術および研究アプリケーションに、より実用的かつ適切になった。

An important issue in medical imaging is the ability to access the processed data quickly and easily.

医療画像の重要な問題は、処理されたデータにすばやく簡単にアクセスできることだ。

Web-based soft- ware tools are becoming a popular solution [22–30].

Web ベースのソフトウェアツールは、一般的なソリューションになりつつある[22-30]。

One such example is the Web-Based Multi-layer Visualization System (WBMVS) [30].

このような例の 1 つに Web-Based Multi-layer Visualization System (WBMVS) [30]がある。

However, these web applications either suffer from poor user-interface (UI) functionalities or require the user to install ActiveX or Java Applet components.

ただし、これらの Web アプリケーションは、ユーザーインターフェイス (UI) の機能が不十分であるか、ユーザーに ActiveX または Java アプレットコンポーネントをインストールする必要がある。

This causes the loss of many advantages of web-based applications.

これにより、Web ベースのアプリケーションの多くの利点が失われる。

Using a web application, by centralized storage of data and availability of the software on different platforms, physicians can easily access patient's data from any computer station, regardless of relative processing power or operating system.

Web アプリケーションを使用することで、データを集中的に格納し、異なるプラットフォーム上でのソフトウェアの可用性を発揮できるため、医師は相対的な処理能力やオペレーティングシステムに関係なく、どのコンピュータステーションからでも患者のデータに簡単にアクセスできる。

There are some fundamental advantages in using web-based medical software.

Web ベースの医療用ソフトウェアの使用にはいくつかの基本的な利点がある。

Because of rapid changes in medical tools, the ability to update the software without interrupting the users and modifying their hardware and local software is necessary.

医療ツールの急速な変化のために、ユーザーを中断せずにハードウェアとローカルソフトウェアを変更することなくソフトウェアを更新する能力が必要だ。

Further, there are many complex processing algorithms that require large memory and computational resources that may not be available on many client machines [30].

さらに、多くのクライアントマシンでは利用できない大量のメモリと計算リソースを必要とする複雑な処理アルゴリズムが多数存在する[30]。

By centralizing processing power and using server machines with large computational resources, it is possible for the client machine to work as a simple terminal.

処理能力を集中化し、大きな計算リソースを持つサーバー・マシンを使用することにより、クライアントマシンは単純な端末として機能することができる。

The acquired data can be saved on the server side and users can access them via the network.

取得したデータはサーバー側に保存することができ、ユーザーはネットワーク経由でアクセスすること

ができる。

The importance of this structure is particularly evident in emergency conditions where quick decisions should be made in locations where resources are limited, such as outside of the hospital or in the field.

この構造の重要性は、病院の外や現場など、限られた場所で迅速な決定が行われるような緊急時に特に顕著だ。

Physicians can access and process the patient's medical data quickly and easily using the proposed web-based medical imaging software.

医師は、提案されたウェブベースの医療画像ソフトウェアを使用して、迅速かつ容易に患者の医療データにアクセスし、処理することができる。

Although there are many Picture Archiving and Communication Systems (PACS) available for centralized data access, most of these systems are designed for Local Area Network (LAN) access, not for internet use.

集中データアクセスのために利用できる PACS (Picture Archiving and Communication System) は数多くありますが、これらのシステムのほとんどは、インターネットではなくローカルエリアネットワーク (LAN) アクセス用に設計されている。

Also, all current web-based PACS systems are based on ActiveX or Java Applet, which are not considered pure web-based solutions, because they impose pre-installation requisites and are not platform independent.

また、現在の Web ベースの PACS システムはすべて、ActiveX や Java アプレットをベースにしています。これらのシステムは、プリインストール要件を満たし、プラットフォームに依存しないため、純粋な Web ベースのソリューションとは見なされません。

In addition, current PACS solutions normally do not perform algorithmic processing on the image and simply serve as image viewers.

さらに、現在の PACS ソリューションは、通常、画像に対してアルゴリズム処理を実行せず、単に画像ビューアとして機能する。

In this paper, we introduce a pure web-based, interactive, extendable, 3D and 2D medical image processing and visualization application, which requires no client installation.

本論文では、クライアント側のインストールを必要としない、Web ベースのインタラクティブで拡張可能な 3D および 2D の医療画像処理およびビジュアライゼーションアプリケーションを紹介する。

Our software uses a four-layered design consisting of an algorithm layer, web-UI layer, server communication layer, and wrapper layer.

我々のソフトウェアは、アルゴリズム層、Web-UI 層、サーバー通信層、およびラップ層からなる 4 層設

計を使用している。

To compete with extendibility of the current local medical image processing software, each layer is highly independent of the other layers.

現行のローカル医療画像処理ソフトウェアの拡張性と競合するために、各層は他の層とは独立している。

A wide range of medical image preprocessing, registration, and segmentation methods are implemented using open source libraries.

幅広い医療画像の前処理、登録、およびセグメンテーションの方法は、オープンソースライブラリを使用して実装されている。

**Desktop-like user interaction is provided by using AJAX technology in the Web- UI.**

Web UI で AJAX テクノロジを使用することによって、デスクトップのようなユーザー対話が提供される。

For the visualization functionality of the software, the VRML standard is used to provide 3D features over the web.  
ソフトウェアの視覚化機能のために、VRML 標準を使用して Web 上に 3D 機能を提供する。

**Integration of these technologies has allowed implementation of our purely web-based software with high functionality without requiring powerful computational resources in the client side.**

これらのテクノロジーを統合することで、クライアント側で強力な計算リソースを必要とせず、純粹に高機能な Web ベースのソフトウェアを実装することができた。

**The UI is designed such that the users can select appropriate parameters for practical research and clinical studies.**

UI は、ユーザーが実際の研究や臨床研究に適したパラメータを選択できるように設計されている。

**The organization of the rest of the paper is as follows.**

残りの論文の構成は次のとおりだ。

**In Section 2, we describe design constraints and technology choices.**

第2節では、設計制約と技術選択について述べる。

**Section 3 describes our software architecture.**

セクション3では、当社のソフトウェアアーキテクチャについて説明する。

**Section 4 describes the user-interface of the software.**

セクション4では、ソフトウェアのユーザーインターフェイスについて説明する。

Section 5 compares our software with software tools developed previously for representing, processing, and visualization of medical images.

5章では、私たちのソフトウェアを、医療画像の表示、処理、視覚化のために以前に開発されたソフトウェアツールと比較する。

# Garbage-First Garbage Collection

David Detlefs, Christine Flood, Steve Heller, Tony Printezis  
Sun Microsystems, Inc.

1 Network Drive, Burlington, MA 01803, USA

{david.detlefs, christine.flood, steve.heller, tony.printezis}@sun.com

## ABSTRACT

*Garbage-First* is a server-style garbage collector, targeted for multi-processors with large memories, that meets a soft real-time goal with high probability, while achieving high throughput. Whole-heap operations, such as global marking, are performed concurrently with mutation, to prevent interruptions proportional to heap or live-data size. Concurrent marking both provides collection “completeness” and identifies regions ripe for reclamation via compacting evacuation. This evacuation is performed in parallel on multi-processors, to increase throughput.

### Categories and Subject Descriptors:

D.3.4 [Programming Languages]: Processors—*Memory management (garbage collection)*

**General Terms:** Languages, Management, Measurement, Performance

**Keywords:** concurrent garbage collection, garbage collection, garbage-first garbage collection, parallel garbage collection, soft real-time garbage collection

## 1. INTRODUCTION

The Java™ programming language is widely used in large server applications. These applications are characterized by large amounts of live heap data and considerable thread-level parallelism, and are often run on high-end multiprocessors. Throughput is clearly important for such applications, but they often also have moderately stringent (though *soft*) real-time constraints, e.g. in telecommunications, call-processing applications (several of which are now implemented in the Java language), delays of more than a fraction of a second in setting up calls are likely to annoy customers.

The Java language specification mandates some form of *garbage collection* to reclaim unused storage. Traditional “stop-world” collector implementations will affect an application’s responsiveness, so some form of concurrent and/or incremental collector is necessary. In such collectors, lower pause times generally come at a cost in throughput. Therefore, we allow users to specify a *soft real-time goal*, stating their desire that collection consume no more than  $x$  ms of

any  $y$  ms *time slice*. By making this goal explicit, the collector can try to keep collection pauses as small and infrequent as necessary for the application, but not so low as to decrease throughput or increase footprint unnecessarily. This paper describes the *Garbage-First* collection algorithm, which attempts to satisfy such a soft real-time goal while maintaining high throughput for programs with large heaps and high allocation rates, running on large multi-processor machines.

The Garbage-First collector achieves these goals via several techniques. The heap is partitioned into a set of equal-sized *heap regions*, much like the *train cars* of the Mature-Object Space collector of Hudson and Moss [22]. However, whereas the remembered sets of the Mature-Object Space collector are *unidirectional*, recording pointers from older regions to younger but not vice versa, Garbage-First remembered sets record pointers from all regions (with some exceptions, described in sections 2.4 and 4.6). Recording all references allows an arbitrary set of heap regions to be chosen for collection. A concurrent thread processes log records created by special mutator write barriers to keep remembered sets up-to-date, allowing shorter collections.

Garbage-First uses a snapshot-at-the-beginning (henceforth *SATB*) concurrent marking algorithm [36]. This provides periodic analysis of global reachability, providing completeness, the property that all garbage is eventually identified. The concurrent marker also counts the amount of live data in each heap region. This information informs the choice of which regions are collected: regions that have little live data and more garbage yield more efficient collection, hence the name “Garbage-First”. The SATB marking algorithm also has very small pause times.

Garbage-First employs a novel mechanism to attempt to achieve the real-time goal. Recent hard real-time collectors [4, 20] have satisfied real-time constraints by making collection interruptible at the granularity of copying individual objects, at some time and space overhead. In contrast, Garbage-First copies objects at the coarser granularity of heap regions. The collector has a reasonably accurate model of the cost of collecting a particular heap region, as a function of quickly-measured properties of the region. Thus, the collector can choose a set of regions that can be collected within a given pause time limit (with high probability). Further, collection is delayed if necessary (and possible) to avoid violating the real-time goal. Our belief is that abandoning hard real-time guarantees for this softer best-effort style may yield better throughput and space usage, an appropriate tradeoff for many applications.

## 2. DATA STRUCTURES/MECHANISMS

In this section, we describe the data structures and mechanisms used by the Garbage-First collector.

### 2.1 Heap Layout/Heap Regions/Allocation

The Garbage-First heap is divided into equal-sized *heap regions*, each a contiguous range of virtual memory. Allocation in a heap region consists of incrementing a boundary, *top*, between allocated and unallocated space. One region is the *current allocation region* from which storage is being allocated. Since we are mainly concerned with multi-processors, mutator threads allocate only *thread-local allocation buffers*, or *TLABs*, directly in this heap region, using a *compare-and-swap*, or CAS, operation. They then allocate objects privately within those buffers, to minimize allocation contention. When the current allocation region is filled, a new allocation region is chosen. Empty regions are organized into a linked list to make region allocation a constant time operation.

Larger objects may be allocated directly in the current allocation region, outside of TLABs. Objects whose size exceeds 3/4 of the heap region size, however, are termed *humongous*. Humongous objects are allocated in dedicated (contiguous sequences of) heap regions; these regions contain only the humongous object.<sup>1</sup>

### 2.2 Remembered Set Maintenance

Each region has an associated *remembered set*, which indicates all locations that might contain pointers to (live) objects within the region. Maintaining these remembered sets requires that mutator threads inform the collector when they make pointer modifications that might create inter-region pointers. This notification uses a *card table* [21]: every 512-byte *card* in the heap maps to a one-byte entry in the card table. Each thread has an associated *remembered set log*, a current buffer or sequence of modified cards. In addition, there is a global set of *filled RS buffers*.

The remembered sets themselves are sets (represented by hash tables) of cards. Actually, because of parallelism, each region has an associated array of several such hash tables, one per parallel GC thread, to allow these threads to update remembered sets without interference. The logical contents of the remembered set is the union of the sets represented by each of the component hash tables.

The remembered set write barrier is performed after the pointer write. If the code performs the pointer write `x.f = y`, and registers `rX` and `rY` contain the object pointer values `x` and `y` respectively, then the pseudo-code for the barrier is:

```

1| rTmp := rX XOR rY
2| rTmp := rTmp >> LogOfHeapRegionSize
3| // Below is a conditional move instr
4| rTmp := (rY == NULL) then 0 else rTmp
5| if (rTmp == 0) goto filtered
6| call rs_enqueue(rX)
7| filtered:

```

This barrier uses a filtering technique mentioned briefly by Stefanović *et al.* in [32]. If the write creates a pointer from an object to another object in the same heap region, a case we expect to be common, then it need not be recorded in a remembered set. The exclusive-or and shifts of lines 1 and 2

<sup>1</sup>Humongous objects complicate the system in various ways. We will not cover these complications in this paper.

means that `rTmp` is zero after the second line if `x` and `y` are in the same heap region. Line 4 adds filtering of stores of null pointers. If the store passes these filtering checks, then it creates an out-of-region pointer. The *rs\_enqueue* routine reads the card table entry for the object head `rX`. If that entry is already dirty, nothing is done. This reduces work for multiple stores to the same card, a common case because of initializing writes. If the card table entry is not dirty, then it is dirtied, and a pointer to the card is enqueued on the thread's remembered set log. If this enqueue fills the thread's current log buffer (which holds 256 elements by default), then that buffer is put in the global set of filled buffers, and a new empty buffer is allocated.

The concurrent remembered set thread waits (on a condition variable) for the size of the filled RS buffer set to reach a configurable *initiating* threshold (the default is 5 buffers). The remembered set thread processes the filled buffers as a queue, until the length of the queue decreases to 1/4 of the initiating threshold. For each buffer, it processes each card table pointer entry. Some cards are *hot*: they contain locations that are written to frequently. To avoid processing hot cards repeatedly, we try to identify the hottest cards, and defer their processing until the next evacuation pause (see section 2.3 for a description of evacuation pauses). We accomplish this with a second card table that records the number of times the card has been dirtied since the last evacuation pause (during which this table, like the card table proper, is cleared). When we process a card we increment its count in this table. If the count exceeds a *hotness threshold* (default 4), then the card is added to circular buffer called the *hot queue* (of default size 1 K). This queue is processed like a log buffer at the start of each evacuation pause, so it is empty at the end. If the circular buffer is full, then a card is evicted from the other end and processed.

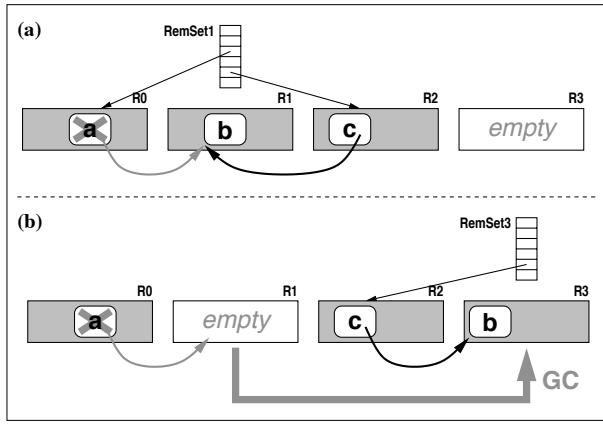
Thus, the concurrent remembered set thread processes a card if it has not yet reached the hotness threshold, or if it is evicted from the hot queue. To process a card, the thread first resets the corresponding card table entry to the *clean* value, so that any concurrent modifications to objects on the card will redirty and re-enqueue the card.<sup>2</sup> It then examines the pointer fields of all the objects whose modification may have dirtied the card, looking for pointers outside the containing heap region. If such a pointer is found, the card is inserted into the remembered set of the referenced region.

We use only a single concurrent remembered set thread, to introduce parallelism when idle processors exist. However, if this thread is not sufficient to service the rate of mutation, the filled RS buffer set will grow too large. We limit the size of this set; mutator threads attempting to add further buffers perform the remembered set processing themselves.

### 2.3 Evacuation Pauses

At appropriate points (described in section 3.4), we stop the mutator threads and perform an evacuation pause. Here we choose a *collection set* of regions, and evacuate the regions by copying all their live objects to other locations in the heap, thus freeing the collection set regions. Evacuation pauses exist to allow compaction: object movement must appear atomic to mutators. This atomicity is costly to achieve in truly concurrent systems, so we move objects during incremental stop-world pauses instead.

<sup>2</sup>On non-sequentially consistent architectures memory barriers may be necessary to prevent reorderings.



**Figure 1: Remembered Set Operation**

If a multithreaded program is running on a multiprocessor machine, using a sequential garbage collector can create a performance bottleneck. We therefore strive to parallelize the operations of an evacuation pause as much as possible.

The first step of an evacuation pause is to sequentially choose the collection set (section 3 details the mechanisms and heuristics of this choice). Next, the main parallel phase of the evacuation pause starts. GC threads compete to claim tasks such as scanning pending log buffers to update remembered sets, scanning remembered sets and other root groups for live objects, and evacuating the live objects. There is no explicit synchronization among tasks other than ensuring that each task is performed by only one thread.

The evacuation algorithm is similar to the one reported by Flood *et al.* [18]. To achieve fast parallel allocation we use *GCLABs*, i.e. thread-local GC allocation buffers (similar to mutator TLABs). Threads allocate an object copy in their GCLAB and compete to install a forwarding pointer in the old image. The winner is responsible for copying the object and scanning its contents. A technique based on *work-stealing* [1] provides load balancing.

Figure 1 illustrates the operation of an evacuation pause. Step A shows the remembered set of a collection set region R1 being used to find pointers into the collection set. As will be discussed in section 2.6, pointers from objects identified as garbage via concurrent marking (object *a* in the figure) are not followed.

## 2.4 Generational Garbage-First

Generational garbage collection [34, 26] has several advantages, which a collection strategy ignores at its peril. Newly allocated objects are usually more likely to become garbage than older objects, and newly allocated objects are also more likely to be the target of pointer modifications, if only because of initialization. We can take advantage of both of these properties in Garbage-First in a flexible way. We can heuristically designate a region as *young* when it is chosen as a mutator allocation region. This commits the region to be a member of the next collection set. In return for this loss of heuristic flexibility, we gain an important benefit: remembered set processing is not required to consider modifications in young regions. Reachable young objects will be scanned after they are evacuated as a normal part of the next evacuation pause.

Note that a collection set can contain a mix of young and

non-young regions. Other than the special treatment for remembered sets described above, both kinds of regions are treated uniformly.

Garbage-First runs in two modes: generational and *pure garbage-first*. Generational mode is the default, and is used for all performance results in this paper.

There are two further “submodes” of generational mode: evacuation pauses can be *fully* or *partially* young. A fully-young pause adds all (and only) the allocated young regions to the collection set. A partially-young pause chooses all the allocated young regions, and may add further non-young regions, as pause times allow (see section 3.2.1).

## 2.5 Concurrent Marking

Concurrent marking is an important component of the system. It provides collector completeness without imposing any order on region choice for collection sets (as, for example, the Train algorithm of Hudson and Moss [22] does). Further, it provides the live data information that allows regions to be collected in “garbage-first” order. This section describes our concurrent marking algorithm.

We use a form of snapshot-at-the-beginning concurrent marking [36]. In this style, marking is guaranteed to identify garbage objects that exist at the start of marking, by marking a logical “snapshot” of the object graph existing at that point. Objects allocated during marking are necessarily considered live. But while such objects must be considered marked, they need not be traced: they are not part of the object graph that exists at the start of marking. This greatly decreases concurrent marking costs, especially in a system like Garbage-First that has no physically separate young generation treated specially by marking.

### 2.5.1 Marking Data Structures

We maintain two *marking bitmaps*, labeled *previous* and *next*. The previous marking bitmap is the last bitmap in which marking has been completed. The next marking bitmap may be under construction. The two physical bitmaps swap logical roles as marking is completed. Each bitmap contains one bit for each address that can be the start of an object. With the default 8-byte object alignment, this means 1 bitmap bit for every 64 heap bits. We use a *mark stack* to hold (some of) the *gray* (marked but not yet recursively scanned) objects.

### 2.5.2 Initial Marking Pause/Concurrent Marking

The first phase of a marking cycle clears the next marking bitmap. This is performed concurrently. Next, the *initial marking pause* stops all mutator threads, and marks all objects directly reachable from the roots (in the generational mode, initial marking is in fact piggy-backed on a fully-young evacuation pause). Each heap region contains two *top at mark start (TAMS)* variables, one for the previous marking and one for the next. We will refer to these as the *previous* and *next TAMS* variables. These variables are used to identify objects allocated during a marking phase. These objects above a TAMS value are considered implicitly marked with respect to the marking to which the TAMS variable corresponds, but allocation is not slowed down by marking bitmap updates. The initial marking pause iterates over all the regions in the heap, copying the current value of *top* in each region to the next TAMS of that region. Steps A and D of figure 2 illustrate this. Steps B and E of this

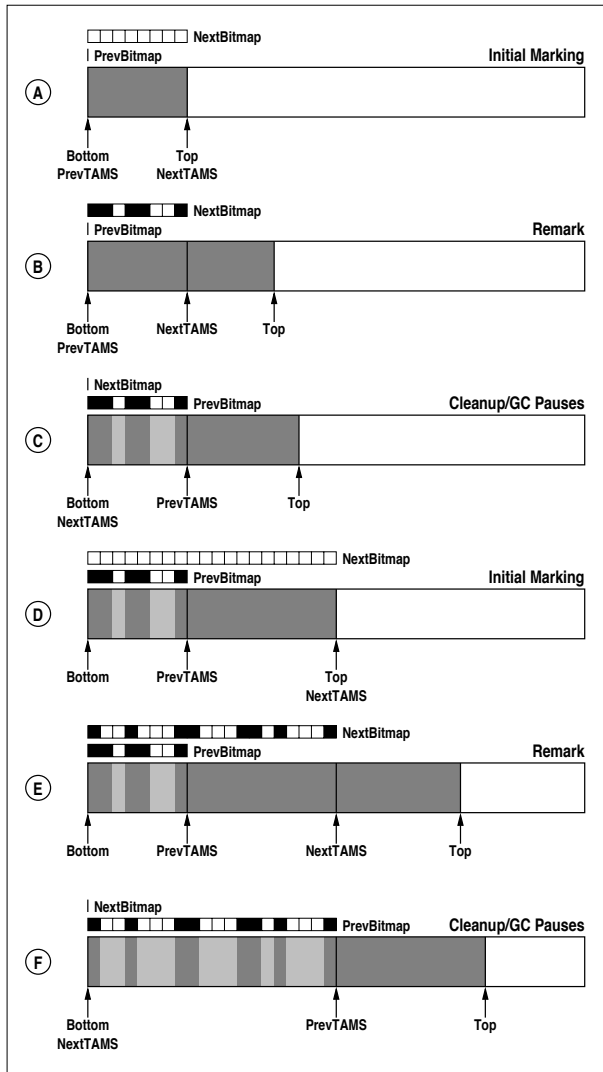


Figure 2: Implicit marking via TAMS variables

figure show that objects allocated during concurrent marking are above the next TAMS value, and are thus considered live. (The bitmaps physically cover the entire heap, but are shown only for the portions of regions for which they are relevant.)

Now mutator threads are restarted, and the concurrent phase of marking begins. This phase is very similar to the concurrent marking phase of [29]: a “finger” pointer iterates over the marked bits. Objects higher than the finger are implicitly gray; gray objects below the finger are represented with a mark stack.

### 2.5.3 Concurrent Marking Write Barrier

The mutator may be updating the pointer graph as the collector is tracing it. This mutation may remove a pointer in the “snapshot” object graph, violating the guarantee on which SATB marking is based. Therefore, SATB marking requires mutator threads to record the values of pointer fields before they are overwritten. Below we show pseudo-code for the marking write barrier for a write of the value in  $rY$  to offset  $FieldOffset$  in an object whose address is in  $rX$ . Its operation is explained below.

```

1| rTmp := load(rThread + MarkingInProgressOffset)
2| if (!rTmp) goto filtered
3| rTmp := load(rX + FieldOffset)
4| if (rTmp == null) goto filtered
5| call satb_enqueue(rTmp)
6| filtered:

```

The actual pointer store  $[rX, FieldOffset] := rY$  would follow. The first two lines of the barrier skip the remainder if marking is not in progress; for many programs, this filters out a large majority of the dynamically executed barriers. Lines 3 and 4 load the value in the object field, and check whether it is null. It is only necessary to log non-null values. In many programs the majority of pointer writes are initializing writes to previously-null fields, so this further filtering is quite effective.

The `satb_enqueue` operation adds the pointer value to the thread’s current marking buffer. As with remembered set buffers, if the enqueue fills the buffer, it then adds it to the global set of completed marking buffers. The concurrent marking thread checks the size of this set at regular intervals, interrupting its heap traversal to process filled buffers.

### 2.5.4 Final Marking Pause

A marking phase is complete when concurrent marking has traversed all the marked objects and completely drained the mark stack, and when all logged updates have been processed. The former condition is easy to detect; the latter is harder, since mutator threads “own” log buffers until they fill them. The purpose of the stop-world final marking pause is to reach this termination condition reliably, while all mutator threads are stopped. It is very simple: any unprocessed completed log buffers are processed as above, and the partially completed per-thread buffers are processed in the same way. This process is done in parallel, to guard against programs with many mutator threads with partially filled marking log buffers causing long pause times or parallel scaling issues.<sup>3</sup>

### 2.5.5 Live Data Counting and Cleanup

Concurrent marking also counts the amount of marked data in each heap region. Originally, this was done as part of the marking process. However, evacuation pauses that move objects that are live must also update the per-region live data count. When evacuation pauses are performed in parallel, and several threads are evacuating objects to the same region, updating this count consistently can be a source of parallel contention. While a variety of techniques could have ameliorated this scaling problem, updating the count represented a significant portion of evacuation pause cost even with a single thread. Therefore, we opted to perform all live data counting concurrently. When final marking is complete, the GC thread re-examines each region, counting the bytes of marked data below the TAMS value associated with the marking. This is something like a sweeping phase, but note that we find live objects by examining the marking bitmap, rather than by traversing dead objects.

As will be discussed in section 2.6, evacuation pauses occurring during marking may increase the next TAMS value

<sup>3</sup>Note that there is no need to trace again from the roots: we examined the roots in the initial marking pause, marking all objects directly reachable in the original object graph. All objects reachable from the roots after the final marking pause has completed marking must be live with respect to the marking.



of some heap regions. So a final stop-world *cleanup* pause is necessary to reliably finish this counting process. This cleanup phase also completes marking in several other ways. It is here that the next and previous bitmaps swap roles: the newly completed bitmap becomes the previous bitmap, and the old one is available for use in the next marking. In addition, since the marking is complete, the value in the next TAMS field of each region is copied into the previous TAMS field, as shown in steps C and F of figure 2. Liveness queries rely on the previous marking bitmap and the previous TAMS, so the newly-completed marking information will now be used to determine object liveness. In figure 2, light gray indicates objects known to be dead. Steps D and E show how the results of a completed marking may be used while a new marking is in progress.

Finally, the cleanup phase sorts the heap regions by expected *GC efficiency*. This metric divides the marking's estimate of garbage reclaimable by collecting a region by the cost of collecting it. This cost is estimated based on a number of factors, including the estimated cost of evacuating the live data and the cost of traversing the region's remembered set. (Section 3.2.1 discusses our techniques for estimating heap region GC cost.) The result of this sorting is an initial ranking of regions by desirability for inclusion into collection sets. As discussed in section 3.3, the cost estimate can change over time, so this estimate is only initial.

Regions containing no live data whatsoever are immediately reclaimed in this phase. For some programs, this method can reclaim a significant fraction of total garbage.

## 2.6 Evacuation Pauses and Marking

In this section we discuss the two major interactions between evacuation pauses and concurrent marking.

First, an evacuation pause never evacuates an object that was proven dead in the last completed marking pass. Since the object is dead, it obviously is not referenced from the roots, but it might be referenced from other dead objects. References within the collection set are followed only if the referring object is found to be live. References from outside the collection set are identified by the remembered sets; objects identified by the remembered sets are ignored if they have been shown to be dead.

Second, when we evacuate an object during an evacuation pause, we need to ensure that it is marked correctly, if necessary, with respect to both the previous and next markings. It turns out that this is quite subtle and tricky. Unfortunately, due to space restrictions, we cannot give here all the details of this interaction.

We allow evacuation pauses to occur when the marking thread's marking stack is non-empty: if we did not, then marking could delay a desired evacuation pause by an arbitrary amount. The marking stack entries may refer to objects in the collection set. Since these objects are marked in the current marking, they are clearly live with respect to the previous marking, and may be evacuated by the evacuation pause. To ensure that marking stack entries are updated properly, we treat the marking stack as a source of roots.

## 2.7 Popular Object Handling

A *popular* object is one that is referenced from many locations. This section describes special handling for popular objects that achieves two goals: smaller remembered sets and a more efficient remembered set barrier.

We reserve a small prefix of the heap regions to contain popular objects. We attempt to identify popular objects quickly, and isolate them in this prefix, whose regions are never chosen for collection sets.

When we update region remembered sets concurrently, regions whose remembered set sizes have reached a given threshold are scheduled for processing in a *popularity pause*; such growth is often caused by popular objects. The popularity pause first constructs an approximate reference count for each object, then evacuates objects whose count reach an individual object popularity threshold to the regions in the popular prefix; non-popular objects are evacuated to the normal portion of the heap. If no individual popular objects are found, no evacuation is performed, but the per-region threshold is doubled, to prevent a loop of such pauses.

There are two benefits to this treatment of popular objects. Since we do not relocate popular objects once they have been segregated, we do not have to maintain remembered sets for popular object regions. We show in section 4.6 that popular object handling eliminates a majority of remembered set entries for one of our benchmarks. We also save remembered set processing overhead by filtering out pointers to popular objects early on. We modify the step of the remembered set write barrier described in 2.2 that filtered out null pointers to instead do:

```
if (rY < PopObjBoundary) goto filtered
```

This test filters both popular objects and also null pointers (using zero to represent null). Section 4.6 also measures the effectiveness of this filtering.

While popular object handling can be very beneficial, it is optional, and disabled in the performance measurements described in section 4, except for the portion of section 4.6 that explicitly investigates popularity. As discussed that section, popular objects effectively decrease remembered set sizes for some applications, but not for all; this mechanism may be superseded in the future.

## 3. HEURISTICS

In the previous section we defined the mechanisms used in the Garbage-First collector. In this section, we describe the heuristics that control their application.

### 3.1 User Inputs

A basic premise of the Garbage-First collector is that the user specifies two things:

- an upper bound on space usage.
- a *soft real-time goal*, in which the user gives: a *time slice*, and a *max GC time* within a time slice that should be devoted to stop-world garbage collection.

In addition, there is currently a flag indicating whether the collector should use *generational mode* (see section 2.4). In the future, we hope to make this choice dynamically.

When attempting to meet the soft real-time goal, we only take into account the stop-world pauses and ignore any concurrent GC processes. On the relatively large multiprocessors we target, concurrent GC can be considered a fairly evenly distributed "tax" on mutator operation. The soft real-time applications we target determine their utilization requirements by benchmarking, not by program analysis, so the concurrent GC load will be factored into this testing.

## 3.2 Satisfying a Soft Real-Time Goal

The soft real-time goal is treated as a primary constraint. (We should be clear that Garbage-First is not a *hard* real-time collector. We meet the soft real-time goal with high probability, but not with absolute certainty.) Meeting such a goal requires two things: ensuring that individual pauses do not exceed the pause time bound, and scheduling pauses so that only the allowed amount of GC activity occurs in any time slice. Below we discuss techniques for meeting these requirements.

### 3.2.1 Predicting Evacuation Pause Times

To meet a given pause time bound, we carefully choose a collection set that can be collected in the available time. We have a model for the cost of an evacuation pause that can predict the incremental cost of adding a region to the collection set. In generational mode, some number of young regions are “mandatory” members of the collection set. In the fully-young submode, the entire collection set is young. Since the young regions are mandatory, we must predict in advance the number of such regions that will yield a collection of the desired duration. We track the fixed and per-regions costs of fully-young collections via historical averaging, and use these estimates to determine the number of young regions allocated between fully-young evacuation pauses.

In the partially-young mode, we may add further non-young regions if pause times permit. In this and pure garbage-first modes, we stop choosing regions when the “best” remaining one would exceed the pause time bound.

In the latter cases, we model the cost of an evacuation pause with collection set  $cs$  as follows:

$$V(cs) = V_{fixed} + U \cdot d + \sum_{r \in cs} (S \cdot rsSize(r) + C \cdot liveBytes(r))$$

The variables in this expression are as follows:

- $V(cs)$  is the cost of collecting collection set  $cs$ ;
- $V_{fixed}$  represents fixed costs, common to all pauses;
- $U$  is the average cost of scanning a card, and  $d$  is the number of dirty cards that must be scanned to bring remembered sets up-to-date;
- $S$  is the cost of scanning a card from a remembered set for pointers into the collection set, and  $rsSize(r)$  is the number of card entries in  $r$ ’s remembered set; and
- $C$  is the cost per byte of evacuating (and scanning) a live object, and  $liveBytes(r)$  is an estimate of the number of live bytes in region  $r$ .

The parameters  $V_{fixed}$ ,  $U$ ,  $S$ , and  $C$  depend on the algorithm implementation and the host platform, and somewhat on the characteristics of the particular application, but we hope they should be fairly constant within a run. We start with initial estimates of these quantities that lead to conservative pause time estimates, and refine these estimates with direct measurement over the course of execution. To account for variation in application behavior, we measure the standard deviation of the sequences of measurements for each parameter, and allow the user to input a further *confidence* parameter, which adjusts the assumed value for

the constant parameter by a given number of standard deviations (in the conservative direction). Obviously, increasing the confidence parameter may decrease the throughput.

The remaining parameters  $d$ ,  $rsSize(r)$ , and  $liveBytes(r)$  are all quantities that can be calculated (or at least estimated) efficiently at the start of an evacuation pause. For  $liveBytes(r)$ , if a region contained allocated objects when the last concurrent marking cycle started, this marking provides an upper bound on the number of live bytes in the region. We (conservatively) use this upper bound as an estimate. For regions allocated since the last marking, we keep a dynamic estimate of the survival rates of recently allocated regions, and use this rate to compute the expected number of live bytes. As above, we track the variance of survival rates, and adjust the estimate according to the input confidence parameter.

We have less control over the duration of the stop-world pauses associated with concurrent marking. We strive therefore to make these as short as possible, to minimize the extent to which they limit the real-time specifications achievable. Section 4.3 shows that we are largely successful and marking-related pauses are quite short.

### 3.2.2 Scheduling Pauses to Meet a Real-Time Goal

Above we have shown how we meet a desired pause time. The second half of meeting a real-time constraint is keeping the GC time in a time slice from exceeding the allowed limit. An important property of our algorithm is that as long as there is sufficient space, we can always delay collection activity: we can delay any of the stop-world phases of marking, and we can delay evacuation pauses, expanding the heap as necessary, at least until the maximum heap size is reached. When a desired young-generation evacuation pause in generational mode must be postponed, we can allow further mutator allocation in regions not designated young, to avoid exceeding the pause time bound in the subsequent evacuation pause.

This scheduling is achieved by maintaining a queue of start/stop time pairs for pauses that have occurred in the most recent time slice, along with the total stop world time in that time slice. It is easy to insert pauses at one end of this queue (which updates the start of the most recent time slice, and may cause pauses at the other end to be deleted as now-irrelevant). With this data structure, we can efficiently answer two forms of query:

- What is the longest pause that can be started now without violating the real-time constraint?
- What is the earliest point in the future at which a pause of a given duration may be started?

We use these primitives to decide how long to delay activities that other heuristics described below would schedule.

## 3.3 Collection Set Choice

This section describes the order in which we consider (non-young, “discretionary”) regions for addition to the collection set in partially-young pauses. As in concurrent marking (section 2.5.5), let the expected GC efficiency of a region be the estimated amount of garbage in it divided by the estimated cost of collecting it. We estimate the garbage using the same  $liveBytes(r)$  calculation we used in estimating the cost. Note that a region may have a small amount of

live data, yet still have low estimated efficiency because of a large remembered set. Our plan is to consider the regions in order of decreasing estimated efficiency.

At the end of marking we sort all regions containing marked objects according to efficiency. However, this plan is complicated somewhat by the fact that the cost of collecting a region may change over time: in particular, the region’s remembered set may grow. So this initial sorting is considered approximate. At the start of an evacuation pause, a fixed-size prefix of the remaining available regions, ordered according to this initial efficiency estimate, is resorted according to the current efficiency estimate, and then the regions in this new sorting are considered in efficiency order.

When picking regions, we stop when the pause time limit would be exceeded, or when the surviving data is likely to exceed the space available. However, we do have a mechanism for handling evacuation failure when necessary.

### 3.4 Evacuation Pause Initiation

The evacuation pause initiation heuristics differ significantly between the generational and pure garbage-first modes.

First, in all modes we choose a fraction  $h$  of the total heap size  $M$ : we call  $h$  the *hard margin*, and  $H = (1 - h)M$  the *hard limit*. Since we use evacuation to reclaim space, we must ensure that there is sufficient “to-space” to evacuate into; the hard margin ensures that this space exists. Therefore, when allocated space reaches the hard limit an evacuation pause is always initiated, even if doing so would violate the soft real-time goal. Currently  $h$  is a constant but, in the future, it should be dynamically adjusted. E.g., if we know the maximum pause duration  $P$ , and have an estimate of the per-byte copying cost  $C$ , we can calculate the maximum “to-space” that could be copied into in the available time.

In fully-young generational mode, we maintain a dynamic estimate of the number of young-generation regions that leads to an evacuation pause that meets the pause time bound, and initiate a pause whenever this number of young regions is allocated. For steady-state applications, this leads to a natural period between evacuation pauses. Note that we can meet the soft real-time goal only if this period exceeds its time slice. In partially-young mode, on the other hand, we do evacuation pauses as often as the soft real-time goal allows. Doing pauses at the maximum allowed frequency minimizes the number of young regions collected in those pauses, and therefore maximizes the number of non-young regions that may be added to the collection set.

A generational execution starts in fully-young mode. After a concurrent marking pass is complete, we switch to partially-young mode, to harvest any attractive non-young regions identified by the marking. We monitor the efficiency of collection; when the efficiency of the partial collections declines to the efficiency of fully-young collections, we switch back to fully-young mode. This rule is modified somewhat by a factor that reflects the heap occupancy: if the heap is nearly full, we continue partially-young collections even after their efficiency declines. The extra GC work performed decreases the heap occupancy.

### 3.5 Concurrent Marking Initiation

In generational mode, our heuristic for triggering concurrent marking is simple. We define a second *soft margin*  $u$ , and call  $H - uM$  the *soft limit*. If the heap occupancy ex-

ceeds the soft limit before an evacuation pause, then marking is initiated as soon after the pause as the soft real-time goal allows. As with the hard margin, the soft margin is presently a constant, but will be calculated dynamically in the future. The goal in sizing the soft margin is to allow concurrent marking to complete without reaching the hard margin (where collections can violate the soft real-time goal).

In pure garbage-first mode, the interaction between evacuation pause and marking initiation is more interesting. We attempt to maximize the *cumulative efficiency* of sequences consisting of a marking cycle and subsequent evacuation pauses that benefit from the information the marking provides. The marking itself may collect some completely empty regions, but at considerable cost. The first collections after marking collect very desirable regions, making the cumulative efficiency of the marking and collections rise. Later pauses, however, collect less desirable regions, and the cumulative efficiency reaches a maximum and begins to decline. At this point we initiate a new marking cycle: in a steady-state program, all marking sequences will be similar, and the overall collection efficiency of the execution will be the same as that of the individual sequences, which have been maximized.

## 4. PERFORMANCE EVALUATION

We have implemented Garbage-First as part of a pre-1.5-release version of the Java HotSpot Virtual Machine. We ran on a Sun V880, which has 8 750 MHz UltraSPARC™ III processors. We used the Solaris™ 10 operating environment. We use the “client” Java system; it was easier to modify the simpler client compiler to emit our modified write barriers.

We sometimes compare with the *ParNew + CMS* configuration of the production Java HotSpot VM, which couples a parallel copying young-generation collector with a concurrent mark-sweep old generation [29]. This is the Java HotSpot VM’s best configuration for applications that require low pause times, and is widely used by our customers.

### 4.1 Benchmark Applications

We use the following two benchmark applications:

- **SPECjbb.** This is intended to model a business-oriented object database. We run with 8 warehouses, and report throughput and maximum transaction times. In this configuration its maximum live data size is approximately 165 M.
- **telco.** A benchmark, based on a commercial product, provided to exercise a telephone call-processing application. It requires a maximum 500 ms latency in call setup, and determines the maximum throughput (measured in calls/sec) that a system can support. Its maximum live data size is approximately 100 M.

For all runs we used 8 parallel GC threads and a confidence parameter of  $1\sigma$ . We deliberately chose not to use the SPECjvm98 benchmark suite. The Garbage-First collector has been designed for heavily multi-threaded applications with large heap sizes and these benchmarks have neither attribute. Additionally, we did not run the benchmarks with a non-incremental collector; long GC pauses cause **telco** to time out and halt.

Benchmark/ configuration	Soft real-time goal compliance statistics by Heap Size								
	V%	avgV%	wV%	V%	avgV%	wV%	V%	avgV%	wV%
<b>SPECjbb</b>	<b>512 M</b>			<b>640 M</b>			<b>768 M</b>		
<b>G-F (100/200)</b>	4.29%	36.40%	100.00%	1.73%	12.83%	63.31%	1.68%	10.94%	69.67%
<b>G-F (150/300)</b>	1.20%	5.95%	15.29%	1.51%	4.01%	20.80%	1.78%	3.38%	8.96%
<b>G-F (150/450)</b>	1.63%	4.40%	14.32%	3.14%	2.34%	6.53%	1.23%	1.53%	3.28%
<b>G-F (150/600)</b>	2.63%	2.90%	5.38%	3.66%	2.45%	8.39%	2.09%	2.54%	8.65%
<b>G-F (200/800)</b>	0.00%	0.00%	0.00%	0.34%	0.72%	0.72%	0.00%	0.00%	0.00%
<b>CMS (150/450)</b>	23.93%	82.14%	100.00%	13.44%	67.72%	100.00%	5.72%	28.19%	100.00%
<b>Telco</b>	<b>384 M</b>			<b>512 M</b>			<b>640 M</b>		
<b>G-F (50/100)</b>	0.34%	8.92%	35.48%	0.16%	9.09%	48.08%	0.11%	12.10%	38.57%
<b>G-F (75/150)</b>	0.08%	11.90%	19.99%	0.08%	5.60%	7.47%	0.19%	3.81%	9.15%
<b>G-F (75/225)</b>	0.44%	2.90%	10.45%	0.15%	3.31%	3.74%	0.50%	1.04%	2.07%
<b>G-F (75/300)</b>	0.65%	2.55%	8.76%	0.42%	0.57%	1.07%	0.63%	1.07%	2.91%
<b>G-F (100/400)</b>	0.57%	1.79%	6.04%	0.29%	0.37%	0.54%	0.44%	1.52%	2.73%
<b>CMS (75/225)</b>	0.78%	35.05%	100.00%	0.54%	32.83%	100.00%	0.60%	26.39%	100.00%

Table 1: Compliance with soft real-time goals for SPECjbb and telco

## 4.2 Compliance with the Soft Real-time Goal

In this section we show how successfully we meet the user-defined soft real-time goal (see section 3.1 for an explanation why we only report on stop-world activity and exclude concurrent GC overhead from our measurements). Presenting this data is itself an interesting problem. One option is to calculate the *Minimum Mutator Utilization* (or MMU) curve of each benchmark execution, as defined by Cheng and Blelloch [10]. This is certainly an interesting measurement, especially for applications and collectors with hard real-time constraints. However, its inherently worst-case nature fails to give any insight into how often the application failed to meet its soft real-time goal and by how much.

Table 1 reports three statistics for each soft real-time goal/heap size pair, based on consideration of all the possible time slices of the duration specified in the goal:

- *V%*: the percentage of time slices that are violating, i.e. whose GC time exceeds the max GC time of the soft real-time goal.
- *avgV%*: the average amount by which violating time slices exceed the max GC time, expressed as a percentage of the desired minimum mutator time in a time slice (i.e. time slice minus max GC time), and
- *wV%*: the excess GC time in the worst time slice, i.e. the GC time in the time slice(s) with the most GC time minus max GC time, expressed again as a percentage of the desired minimum mutator time in a time slice.

There are ways to accurately calculate these measurements. However, we chose, instead, to approximate them by quantizing in 1 ms increments. We test five soft real-time goals, for each of three heap sizes, for each benchmark. The statistics shown in the table were gathered from a single benchmark run, as it was not obvious how to combine results from several runs in a useful way. The combination of soft real-time goals that we chose to use allows us to observe how the behavior of Garbage-First changes for different max GC times, but also for a fixed max GC time and different time slices. Rows are labeled with the collector used and the real time goal “(X/Y),” where Y is the time slice, and X the max GC time within the time slice, both in ms. Each group of three columns is devoted to a heap size. We also show the behavior of the Java HotSpot VM’s ParNew +

CMS collector. This collector does not accept an input real-time goal, so we chose a young generation size that gave average young-generation pauses comfortably less than the pause time allowed by one of the Garbage-First real-time goals, then evaluated the ParNew + CMS collector against the chosen goal, which is indicated in the row label.

Considering the Garbage-First configurations, we see that we succeed quite well at meeting the given soft real-time goal. For **telco**, which is a real commercial application, the goal is violated in less than 0.5% of time slices, in most cases, less than 0.7% in all cases, and by small average amounts relative to the allowed pause durations. **SPECjbb** is somewhat less successful, but still restricts violations to under 5% in all cases, around 2% or less in most. Generally speaking, the violating percentage increases as time slice durations increase. On the other hand, the average violation amount and excess GC time in the worst time slices tend to decrease as the time slice durations increase (as a small excess is divided by a larger desired mutator time). Garbage-First struggled with the 100/200 configuration in **SPECjbb**, having a maximal wV% for the 512 M heap size and a wV% of over 60% for the other two heap sizes. It behaved considerably better, however, in the rest of the configurations. In fact, **SPECjbb** with the 200/800 configuration gave the best overall results, causing no violations at all for two of three heap sizes shown and very few in the third one.

The CMS configurations generally have considerably worse behavior in all metrics. For **telco**, the worst-case violations are caused by the final phase of concurrent marking. For **SPECjbb**, this final marking phase also causes maximal violations, but in addition, the smaller heap size induces full (non-parallel) stop-world GCs whose durations are considerably greater than the time slices.

## 4.3 Throughput/GC Overhead

Figures 3 and 4 show application throughput for various heap sizes and soft real-time goals. We also show ParNew + CMS (labeled “CMS”) performance for young-generation sizes selected to yield average collection times (in the application steady state) similar to the pause times of one of the Garbage-First configurations. The **SPECjbb** results shown are averages over three separate runs, the **telco** results are averages over fifteen separate runs (in an attempt to smooth out the high variation that the **telco** results exhibit).

In the case of the **SPECjbb** results, CMS actually forced

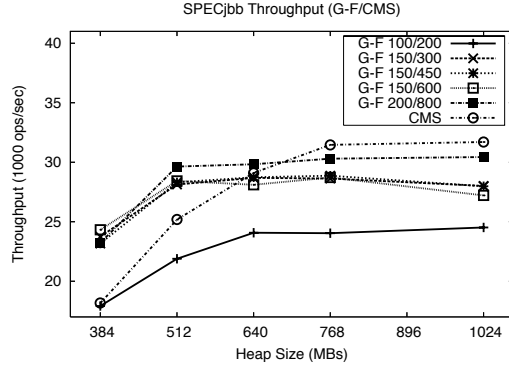


Figure 3: Throughput measurements for SPECjbb

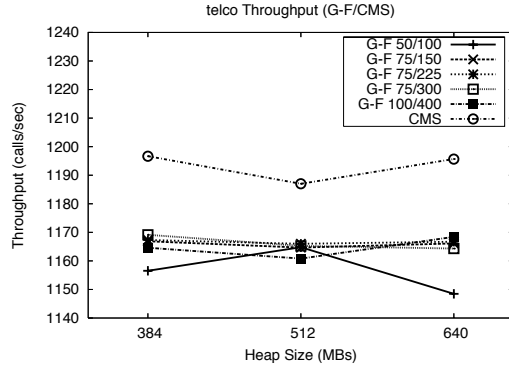


Figure 4: Throughput measurements for telco

full heap collections in heap sizes of 640 M and below (hence, the decrease in throughput for those heap sizes). Garbage-First only reverted to full heap collections in the 384 M case. For the larger heap sizes, Garbage-First seems to have very consistent behaviour, with the throughput being only slightly affected by the heap size. In the Garbage-First configurations, it is clear that increasing the max GC time improves throughput, but varying the time slice for a fixed max GC time has little effect on throughput. Comparing the **SPECjbb** results of Garbage-First and CMS we see that CMS is ahead, by between 5% and 10%.

In the **telco** throughput graph, no configuration of either collector forced full heap collections. The Garbage-First results seem to be affected very little by the max GC time/time slice pairs, apart from the 50/100 configuration that seems to perform slightly worse than all the rest. Again, the comparison between the two collectors shows CMS to be slightly ahead (by only 3% to 4% this time).

Table 2 compares final marking pauses for the Garbage-First and CMS collectors, taking the maximum pauses over all configurations shown in table 1. The results show that the SATB marking algorithm is quite effective at reducing pause times due to marking.

We wish to emphasize that many customers are quite willing to trade off some amount of throughput for more predictable soft real-time goal compliance.

#### 4.4 Parallel Scaling

In this section we quantify how well the collector parallelizes stop-world collection activities. The major such ac-

benchmark	G-F	CMS
<b>SPECjbb</b>	25.4 ms	934.7 ms
<b>telco</b>	48.7 ms	381.7 ms

Table 2: Maximum final marking pauses

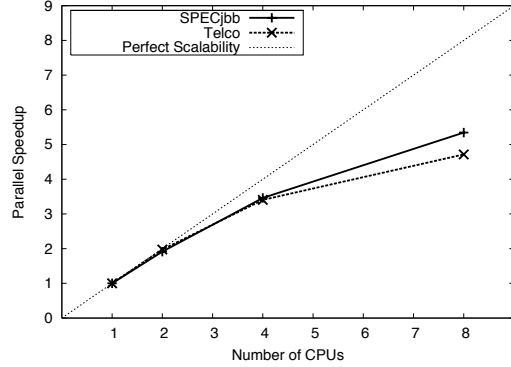


Figure 5: Parallel Scaling of Fully-Young Collections

tivities are evacuation pauses. The adaptive nature of the collector makes this a difficult property to measure: if we double the number of processors available for GC, and this increases the GC rate, then we'll do more collection at each pause, changing the execution significantly. Therefore, for this experiment, we restrict the collector to fully young collections, use a fixed “young generation” size, and use a total heap size large enough to make “old generation” collection activity unnecessary. This measures the scaling at least of fully young collections. We define the *parallel speedup* for  $n$  processors as the total GC time using one parallel thread divided by the total GC time using  $n$  parallel threads (on a machine with least  $n$  processors). This is the factor by which parallelization decreased GC time. Figure 5 graphs parallel speedup as a function of the number of processors.

While these measurements indicate appreciable scaling, we are hoping, with some tuning, to achieve nearly linear scaling in the future (on such moderate numbers of CPUs).

#### 4.5 Space Overhead

The card table has one byte for every 512 heap bytes, or about 0.2% overhead. The second card table used to track card hotness adds a similar overhead (though a smaller table could be used). The two marking bitmaps each have one bit for every 64 heap bits; together this is about a 3% overhead. Parallel task queues are constant-sized; using queues with 32K 4-byte entries per parallel thread avoids task queue overflow on all tests described in this paper. The marking stack is currently very conservatively sized at 8 M, but could be made significantly smaller with the addition of a “restart” mechanism to deal with stack overflow.

Next we consider per heap region overhead. The size of the data structure that represents a region is a negligible fraction of the region size, which is 1 M by default. Heap region remembered sets are more important source of per-region space overhead; we discuss this in the next section.

#### 4.6 Remembered Set Overhead/Popularity

Here, we document the overheads due to remembered set maintenance. We also show how the popular object handling

benchmark	heap size	Rem set space		# pop. pauses	#/bytes pop. objects
		pop. obj handling	no pop. obj handling		
<b>SPECjbb</b>	1024 M	205 M (20.5%)	289 M (28.9%)	3	6/256
<b>telco</b>	500 M	4.8 M (1.0%)	20.4 M (4.0%)	6	60/1736

Table 3: Remembered set space overhead

benchmark	all tests	same region	null pointer	popular object
<b>SPECjbb</b>	72.2	39.9	30.0	2.3
<b>telco</b>	81.7	36.8	37.0	7.9

Table 4: Effectiveness of Remembered Set Filtering

(see section 2.7) can have several positive effects.

Table 3 shows the remembered set overheads. For each benchmark, we show the heap size and space devoted to remembered sets with and without popular object handling. We also show the number of popularity pauses, and the number of popular objects they identify and isolate.

The **telco** results prove that popular object handling can significantly decrease the remembered set space overhead. Clearly, however, the **SPECjbb** results represent a significant problem: the remembered set space overhead is similar to the total live data size. Many medium-lived objects point to long-lived objects in other regions, so these references are recorded in the remembered sets of the long-lived regions. Whereas popularity handling for **telco** identifies a small number of highly popular objects, **SPECjbb** has both a small number of highly popular objects, and a larger number of objects with moderate reference counts. These characteristics cause some popularity pauses to be abandoned, as no individual objects meet the per-object popularity threshold (see section 2.7); the popularity threshold is thus increased, which accounts for the small number of popularity pauses and objects identified for **SPECjbb**.

We are implementing a strategy for dealing with popular heap regions, based on constraining the heap regions chosen for collection. If heap region B’s remembered set contains many entries from region A, then we can delete these entries, and constrain collection set choice such that A must be collected before or at the same time as B. If B also contains many pointers into A, then adding a similar constraint in the reverse direction joins A and B into an *equivalence class* that must be collected as a unit. We keep track of no remembered set entries between members of an equivalence class. Any constraint on a member of the equivalence class becomes a constraint on the entire class. This approach has several advantages: we can limit the size of our remembered sets, we can handle popular regions becoming unpopular without any special mechanisms, and we can collect regions that heavily reference one another together and save remembered set scanning costs. Other collectors [22, 30] have reduced remembered set costs by choosing an order of collection, but the dynamic scheme we describe attempts to get maximal remembered set footprint reduction with as few constraints as possible.

Next we consider the effectiveness of filtering in remembered set write barriers. Table 4 shows that the effectiveness of the various filtering techniques. Each column is the percent of pointer writes filtered by the given test.

The filtering techniques are fairly effective at avoiding the more expensive out-of-line portion of the remembered set

write barrier. Again, **SPECjbb** is something of a worst case. The most effective technique is detection of intra-region writes. We hope to do some portion of the null pointer filtering statically in the future. Popularity filtering, while less effective, is still important: for **telco**, for example, it filters 30% of the writes not filtered by the other techniques.

We were not able to develop an experimental methodology that we trusted to accurately measure the cost of the remembered set and marking write barriers. This bears further investigation.

## 5. RELATED WORK

The Garbage-First collector borrows techniques from an assortment of earlier garbage collectors.

Baker [5] describes incremental copying collection; later variants include [25, 13, 31]. O’Toole and Nettles [28] describe concurrent *replicating collection*. Later variants include [23] and [10].

A large number of concurrent collectors have also been proposed in the past [14, 15, 16, 2]. Boehm *et al.* [8] describe what we will term *mostly-concurrent* collection. Printezis and Detlefs [29] give a variant of mostly-concurrent collection that gains some synergies by implementation in a generational collector with mutator cooperation. This has been since extended by Ossia *et al.* [27]

Collectors that use parallelism to decrease evacuation pauses have been reported by [17, 18].

A recent literature on *oldest-first* techniques [32, 19, 33, 12] has pointed out that while the *youngest-first* technique of generational collection is effective at removing short-lived objects, there is little evidence that the same models apply to longer-lived objects. In particular, it is often a more effective heuristic to concentrate collection activity on the oldest of the longer-lived objects. Pure Garbage-First has borrowed from these ideas.

The idea of dividing collection of a large heap into incremental collection of smaller portions of the heap extends back (at least) to Bishop’s thesis [7]. Probably the most well-known version of this idea is the *Mature Object Space* (a.k.a. *Train*) algorithm of Hudson and Moss [22]. Garbage-First differs from the Train algorithm in using concurrent marking to provide collection completeness, using bi-directional remembered sets to allow a less constraint choice of collection set, and using marking data to guide the choice of an efficient collection set.

Lang and Dupont [24] described an algorithm that picks, at each marking cycle, a section of the heap to be evacuated to another section kept free for that purpose; this is much like a Garbage-First evacuation pause. However, the marking/compacting phase is neither concurrent nor parallel, and only one region is compacted per global marking.

Ben-Yitzhak *et al.* [6] describe a similar system to that of Lang and Dupont. They augment the collector described in [27] with the ability to choose a sub-region of the heap for compaction, constructing its remembered set during marking, then evacuating it in parallel in a stop-world phase.

While this collector clearly has features similar to Garbage-First, there are also important differences. The region to be evacuated is chosen at the start of the collection cycle, and must be evacuated in a single atomic step. In contrast, Garbage-First (by paying the cost of incremental remembered set maintenance) allows compaction to be performed in a number of smaller steps. Ben-Yitzhak *et al.* do not discuss using information from marking to guide the choice of region to compact. Additionally, they report no attempt to meet a user-defined pause time.

Sachindran and Moss [30] recently proposed Mark-Copy, a somewhat similar scheme to Garbage-First, in which a combination of marking and copying is used to collect and compact the old generation. The main algorithm they describe is neither parallel nor concurrent; its purpose is to decrease the space overhead of copying collection. Remembered sets are constructed during marking; like the Train algorithm, these are unidirectional, and thus require a fixed collection order. They sketch techniques that would make marking and remembered set maintenance more concurrent (and thus more like Garbage-First), but they have implemented only the concurrent remembered set maintenance.

Another interesting research area is that of hard real-time garbage collection. Bacon *et al.* describe the *Metronome* [4], an interesting approach to hard real-time collection. Like Garbage-First, the Metronome collector is an SATB concurrent mark-sweep collector, with compaction performed as necessary, that allows the user to specify a hard real-time constraint. A later paper [3] describes heuristics for when to compact, and which and how many regions to compact; these are similar to our collection set choice heuristics. The major difference is that the Metronome, to guarantee hard real-time behavior, performs compaction in small atomic steps. To enable this fine-grained interruptibility, an extra header word is dedicated as a forwarding pointer, and the mutator must execute a read barrier that always follows these pointers (in the style of Brooks [9]; Henriksson [20] describes another hard real-time collector using this technique). Garbage-First avoids these space and time costs; we incur other space and time costs for remembered set maintenance, and do not offer hard real-time guarantees, but we believe these are the right tradeoffs for many of our target applications with soft real-time requirements. Another distinction is that the Metronome collector is targeted to uniprocessor embedded systems, and hence is incremental but neither concurrent nor parallel.

## 6. CONCLUSIONS AND FUTURE WORK

We have presented Garbage-First, a garbage collector that combines concurrent and parallel collection and is targeted for large multi-processor machines with large memories. The main contributions of Garbage-First are the following.

- It is the first high-performance server-style collector that compacts sufficiently to completely avoid the use of fine-grain free lists [29, 27, 6] for allocation. This considerably simplifies parts of the collector and mostly eliminates potential fragmentation issues.
- It uses novel techniques, based on global marking information and other easily-obtainable metrics, to prioritize regions for collection according to their GC efficiency, rather than simply the amount of live data

they contain, and to choose a collection set to meet a pause time goal.

- It tries to schedule future pauses so that they do not violate the given soft real-time goal, using a data structure that tracks previous pauses. It is more flexible than previous work [4], as it can allow multiple pauses of various durations in a time slice.

We have many ideas for improving Garbage-First in the future. We hope to modify write barriers and remembered set representations to increase the efficiency of remembered set processing. We are investigating static analyses to remove write barriers in some cases. Further heuristic tuning may bring benefits. Finally, we believe that there are ways in which Garbage-First may make better use of compile-time escape analysis [11, 35] than standard generational systems can, and plan to investigate this.

## APPENDIX

### A. ACKNOWLEDGMENTS

We would like to acknowledge the contributions of our many colleagues who have contributed to the Garbage-First project over the last several years: Alex Garthwaite, Ross Knippel, William Clinger, Andy C. King, Matthew Hertz, Nandivada Venkata Krishna, and Jan-Willem Maessen.

### B. TRADEMARKS

Java, HotSpot, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the US and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

### C. REFERENCES

- [1] Arora, Blumofe, and Plaxton. Thread scheduling for multiprogrammed multiprocessors. *MST: Mathematical Systems Theory*, 34, 2001.
- [2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on Sliding views. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.
- [3] David F. Bacon, Perry Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 81–92. ACM Press, 2003.
- [4] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.
- [5] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.
- [6] Ori Ben-Yitzhak, Irit Gofit, Elliot Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In David Detlefs, editor, *ISMM'02 Proceedings of the Third International Symposium on*

- Memory Management*, ACM SIGPLAN Notices, pages 100–105, Berlin, June 2002. ACM Press.
- [7] Peter Bishop. *Computer Systems With a Very Large Address Space and Garbage Collection*. PhD thesis, MIT, May 1977.
  - [8] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In Brent Hailpern, editor, *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 157–164, Toronto, ON, Canada, June 1991. ACM Press.
  - [9] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262. ACM, ACM, August 1984.
  - [10] Perry Cheng and Guy Blleloch. A parallel, real-time garbage collector. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 125–136, N.Y., June 20–22 2001. ACM Press.
  - [11] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, October 1999. ACM Press.
  - [12] William D. Clinger and Lars T. Hansen. Generational garbage collection and the radioactive decay model. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 97–108, 1997.
  - [13] David L. Detlefs. Concurrent garbage collection for C+++. Technical Report CMU-CS-90-119, Carnegie-Mellon University, May 1990.
  - [14] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, November 1978.
  - [15] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 113–123, New York, NY, 1993. ACM.
  - [16] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the Symposium: Portland, Oregon, January 17–21, 1994*, pages 70–83, New York, NY, USA, 1994. ACM Press.
  - [17] Toshio Endo, Kenjiro Taura, and Akinori Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of Supercomputing'97 (CD-ROM)*, San Jose, CA, November 1997. ACM SIGARCH and IEEE. University of Tokyo.
  - [18] Christine H. Flood, David Detlefs, Nir Shavit, and Xiaolan Zhang. Parallel garbage collection for shared memory multiprocessors. In *Proceedings of the Java™ Virtual Machine Research and Technology Symposium*, Monterey, April 2001. USENIX.
  - [19] Lars T. Hansen and William D. Clinger. An experimental study of renewal-older-first garbage collection. In *Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 247–258. ACM Press, 2002.
  - [20] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.
  - [21] Urs Hölzle. A fast write barrier for generational garbage collectors. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.
  - [22] Richard L. Hudson and J. Eliot B. Moss. Incremental collection of mature objects. In Yves Bekkers and Jacques Cohen, editors, *International Workshop on Memory Management*, Lecture Notes in Computer Science, pages 388–403, St. Malo, France, September 1992. Springer-Verlag.
  - [23] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. In *Joint ACM Java Grande — ISCOPE 2001 Conference*, Stanford University, CA, June 2001.
  - [24] B. Lang and F. Dupont. Incremental incrementally compacting garbage collection. In *Proceedings SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques*, pages 253–264. ACM, ACM, June 1987.
  - [25] John R. Ellis; Kai Li; and Andrew W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report 25, Digital Equipment Corporation Systems Research Center, February 1988.
  - [26] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *CACM*, 36(6):419–429, June 1983.
  - [27] Yoav Ossia, Ori Ben-Yitzhak, Irit Gofit, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent gc for servers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 129–140. ACM Press, 2002.
  - [28] James O'Toole and Scott Nettles. Concurrent replicating garbage collection. In *Conference on Lisp and Functional programming*. ACM Press, June 1994.
  - [29] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *Proceedings of the International Symposium on Memory Management*, Minneapolis, Minnesota, October 15–19, 2000.
  - [30] Narendran Sachindran and J. Eliot B. Moss. Mark-copy: fast copying gc with less space overhead. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 326–343. ACM Press, 2003.
  - [31] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In Andreas Paepcke, editor, *OOPSLA'91 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 26(11) of *ACM SIGPLAN Notices*, pages 16–32, Phoenix, Arizona, October 1991. ACM Press.
  - [32] Darko Stefanovic, Matthew Hertz, Stephen M. Blackburn, Kathryn S. McKinley, and J. Eliot B. Moss. Older-first garbage collection in practice: evaluation in a java virtual machine. In *Proceedings of the workshop on memory sytem performance*, pages 25–36. ACM Press, 2002.
  - [33] Darko Stefanovic, Kathryn S. McKinley, and J. Eliot B. Moss. Age-based garbage collection. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34.10 of *ACM Sigplan Notices*, pages 370–381, N. Y., November 1–5 1999. ACM Press.
  - [34] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Notices*, 19(5):157–167, May 1984.
  - [35] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, October 1999. ACM Press.
  - [36] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3):181–198, 1990.