TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Visualizing Time-Dependent Software Artifacts

By
Sergio Moreta

Supervisor:
dr. ir. A.C. Telea

Eindhoven, December 2006

Table of Contents

Table of Contents **2**

1 Introduction **5**

2 Time-dependent Software Data **7**
2.1 Dynamic Memory Allocator . . . 7
2.1.1 Memory Allocator Data Model . . . 8
2.1.2 Log Format . . . 10
2.2 Software Configuration Management Systems . . . 11
2.2.1 Evolution Data Model . . . 12
2.3 Generic Data Model . . . 13

3 Visualization Methods **14**
3.1 Visualization Model . . . 14
3.1.1 Layout Model . . . 15
3.1.2 Rendering Model . . . 17
3.2 Cushioning . . . 20
3.2.1 Parabolic cushions . . . 20
3.2.2 Plateau cushions . . . 21
3.3 Sub-sampling . . . 23
3.3.1 Color Sub-sampling . . . 25
3.3.2 Scalar Sub-sampling . . . 27
3.4 Metric Bar . . . 27
3.5 Hierarchical Agglomerative Clustering . . . 29
3.5.1 Distance Metric . . . 30
3.5.2 Clustering Model . . . 32
3.5.3 Sectioning . . . 37
3.5.4 Cluster Coloring . . . 39
3.5.5 Interleaved Cushioning . . . 39
3.6 User Interaction . . . 40
3.6.1 General Tool Description . . . 40
3.6.2 Tasks . . . 41

4 Applications and Results **45**
4.1 Application A: Dynamic Memory Allocator . . . 45
4.1.1 Patterns of Interest . . . 46
4.2 Memory Allocator Application: Results . . . 48
4.2.1 Phases and Patterns . . . 50
4.2.2 Structured Program Behavior . . . 52
4.2.3 Compactness . . . 54
4.2.4 External Fragmentation . . . 54

4.2.5 Internal Fragmentation 56
4.2.6 Attribute Correlations 60
4.3 Application B: Software Configuration Management System 60
4.4 Software Configuration Management System Application: Results 63
4.4.1 Directory Structure 63
4.4.2 Evolution Patterns 63
4.4.3 Change Patterns 66

5 Conclusions and Further Research **67**
5.1 Further Research 68

Foreword

With this master's thesis, I conclude my study of Computer Science and Engineering at the Technische Universiteit Eindhoven. This thesis presents a set of generic techniques and design principles for the visualization of time-dependent software artifacts. These techniques are also implemented in a tool, built during this research.

Through this foreword, I would like to express my gratitude to my supervisor, Alex Telea, for his professional guidance, support, and most of all, his patience. It is highly appreciated.
I thank my family, especially my mother, and Kristel for their love and support. You guys are my pillars. I also thank Matthijs, my co-student and good friend, for his thorough review of what follows.

Chapter 1

Introduction

Software engineering plays a prominent role within the field of study of computer science. As the size of a software project grows software engineering becomes more and more influential. Each phase of the software engineering process produces a variety of different types of products. The quality of these products is of crucial importance to the phases that follow and in the end the likeliness of success of the entire project. These products, yielded by the software engineering process, are collectively referred to as *software artifacts*. Examples are requirements specifications, architecture and design models, source and executable code (programs), configuration directives, test data, test scripts, process models, project plans, various documentation etc.
According to [Con03], a software artifact is any piece of software (i.e. models/descriptions) developed and used during software development and maintenance.
As stated before, these software artifacts form a cornerstone of a software system and their quality is reflected in the quality of the final product. By performing empirical quality analysis on these artifacts, deficiencies and shortcomings can be discovered at early stages of the software engineering project. This analysis is not limited to the independent analysis of a sequence of snap-shots of a software artifact, but can also analyze how it changes over time. By introducing time as an additional dimension, the effectiveness of a well thought-out analysis increases significantly. The evolution of a software artifact is a software artifact of its own, hence all software artifacts can be analyzed in a time-dependent way.
Due to the amount and nature of the data analyzed, the visualization of this information often leads to a better understanding of it. *Information visualization* is defined in [CMS99] as "the use of computer-supported, interactive, visual representations of abstract data to amplify cognition". This is in contrast to *scientific visualization*, which targets the representation of scientific and often physically based data.
Generally, information visualization amplifies cognition by reducing the load on the human working memory. This is achieved by ([CMS99]):

- Grouping together information that is used together. This can avoid large amounts of search in the working memory of humans
- Using location to group information on a single element leads to reduction in the amounts of search in the working memory
- Visual representation automatically represents a large amount of perceptual inferences that are extremely easy for humans.

Software visualization ([SBP97]), a subfield of information visualization, is concerned with the visual representation of information about software systems based on their structure, size, history or behavior. This thesis presents some software visualization techniques to assist in the analysis of time-dependent software artifacts. Its goal is to gain insight into the evolution of complex and large amounts of time-dependent software data, through the analysis of logs. Such logs are

typically weakly structured datasets consisting of hundreds of thousand of low-level events. The following research questions summarize the challenges encountered when researching methods for extracting the higher level structure from these unstructured logs through visualization:

Which visualization methods and techniques can give insight into the evolution of time-dependent software artifacts? How can these methods be implemented efficiently and effectively in tools?

For the purpose of this research, existing techniques are combined with several new techniques. These are then validated by applying them to some real-life datasets from two highly different types of problems. The first problem addresses the behavior of a memory allocator for a mobile phone. The second problem addresses a software project's code-level evolution. These two highly different fields of application are chosen in order to demonstrate that the techniques developed are generic enough to handle and provide insight in a diverse set of application areas having different data models and target questions. By limiting the research to the two application areas above, the reader is hopefully convinced of its versatility, without straying too much from the actual subject of the thesis.
The remainder of this thesis is organized as follows. Chapter 2 describes the two types of software artifacts mentioned above in detail. Firstly, it elaborates on the internal memory organization addressed by a memory allocator, the log format provided by the application profiler that monitors this allocator and the data model inherent to this log. Secondly, it details the acquisition methods for source code evolution information of a software project and a suitable data model. Chapter 3 starts by presenting the basic layout, a 2-dimensional orthogonal visualization, and some preliminary rendering methods. It subsequently describes a number of both existing and new techniques for enhancing the visualization. These include cushioning ([vWvdW99]), a hierarchical agglomerative clustering method and a set of new techniques including several types of importance-based sub-sampling for displaying elements of subpixel size and interleaved cushioning for improving visual segregating between clusters. In chapter 4 the visualization techniques are applied to the two software artifacts of sections 2.1 and 2.2. At the same time, it presents some of the results obtained. A conclusion and suggestions for further research are given in chapter 5.

Chapter 2

Time-dependent Software Data

This chapter describes two types of time-dependent software artifacts which will be analyzed later on in this thesis. These two models are quite different. Yet, both are visualized later using the same types of techniques, which shows that the same techniques can be utilized to acquire insight into a variety of software artifacts.
The first model originates from the need for software testing, a field with rising importance within software engineering. The software artifact in consideration is data on the behavior of a memory allocator running on a mobile phone. The data describes the allocations and de-allocations and the times at which they were done.
The second model is about the evolution of source code in a software project. In this model, the data describes changes made to the source code at points in time, determined by the author of the changes.

2.1 Dynamic Memory Allocator

Embedded systems play an increasing role of importance in our daily lives. They have evolved from being limited to single-purpose, stand-alone devices to a broad range of frequently interconnectable devices serving multiple purposes. One device which has evolved immensely over the past decade is the mobile phone. Not only has it in general become much smaller, it has taken over some tasks which have mostly nothing to do with making telephone calls. They serve as our address book, alarm clock, music player and personal scheduler, to name a few examples. The operating system needs to dedicate a sufficient chunk of the device's resources, e.g. processor or internal memory, to each of these application programs for them to be able to coexist harmoniously. Furthermore, because of the small dimensions of the device, these resources are usually highly limited. The application programs serve different purposes and consequently have diverse resource requirements and somewhat unpredictable patterns for accessing these resources. Especially the task of allocating fragments of memory to the processes efficiently becomes a difficult task because of these diverse patterns and the limited memory space.
The dynamic memory allocator is the component of the operating system that deals with keeping track of which parts of memory are in use and which parts are free. Processes request and free pieces of memory variable in size and it is the allocator that determines which memory fragment they get assigned. In doing this it should try to keep memory fragmentation to a minimum to avoid wasting memory space.
A few popular memory allocator mechanism types include sequential fits, indexed fits, segregated fits and buddy systems [WJNB95], each having its advantages and disadvantages in different application areas. Some allocators, like the one analyzed in section 4.2, attempts to minimize fragmentation by partitioning the available memory into sections each containing blocks of different sizes. Additionally, a section of memory is reserved for allocation requests larger than

the largest block in the partition. This section accommodates variable-sized allocation requests. For the designers of the operating system of a mobile phone this means that they need to make decisions concerning what specific algorithm to use for their dynamic memory allocator and optionally how to partition the memory space into sections of blocks.
Testing how well the dynamic memory allocator works in practice can be done by instrumenting the allocator to log its operations and then analyze this log containing information on all allocations and de-allocations made by certain applications. As typically many thousands of these allocations and de-allocations are made every second, one can imagine that the log produced by profiling the allocator for a few minutes would already grow quite large. General statistics of the allocations and de-allocations made are easily extracted from these files, but this approach leaves global patterns in the behavior of the memory allocator undiscovered.
The designer(s) of the allocator might be interested in having the following questions answered:

- What processes require a lot of memory?
- How much space is wasted by allocating more memory than required?
- How is fragmentation distributed in memory?
- Are there fragmentation patterns which can be ascribed to the allocator?
- Are there other fragmentation patterns which can be ascribed to the monitored application?
- Which are the largest quasi-compact regions allocated?

Even though the first two of these questions can be answered using general statistics, the latter four are more subtle and require a different approach.
The following sections briefly explain the typical organization of address-space in the mobile device which was the subject of our study. Then the format used in the log is described.

2.1.1 Memory Allocator Data Model

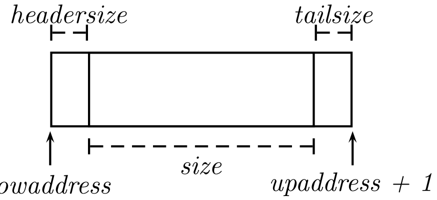
Before diving into the syntactic structure of the allocator profiler log it is important to globally discuss the memory organization of the mobile phone in consideration.
We consider here as typical device, a mobile phone running the SymbianOS operating system. The allocator considered is the one provided by the C runtime library on that platform. It is the piece of code responsible for implementing the **malloc**, **calloc**, **realloc** and **free** operations.
A process running on this system issues a *request* to the allocator. The request contains the size in bytes needed by the process. The allocator then responds to this request by allocating the process a piece of memory of size at least the size of the request.
The memory available to the allocator is partitioned into two sections, namely the pool and the heap. Memory allocated to a process as a response to a request is either allocated in the pool or in the heap depending on the size of the request and the state of the pool. The details of this consideration together with a general description of the pool and heap is presented next.

Pool

The pool is a section of memory with a predefined structure. It keeps groups of blocks of equal size in the same contiguous memory range. These blocks are unable to coalesce into bigger, or split into smaller blocks. There are several such ranges each having a unique block size and holding a predefined number of these blocks.
Stated formally, the pool is subdivided into a set of N bins:

$$Pool = \{Bin_i \mid i \in 1 \ldots N\}$$

Figure 2.1 Composition of a block



Each bin is a contiguous slice of memory and has a lower- and upper-address. In turn, it consists of a predefined number of blocks:

$$Bin_i = \{BL_{i,j} \mid j \in 1 \ldots |Bin_i|\}$$

Each block has a lower-address and a size. Here, size indicates the effective size of the block. As each block also contains a header and a tail holding meta-values for the pool data structure, size is not equal to the amount of memory the block occupies (See figure 2.1).
The block's upper-address follows implicitly from the lower-address, size and the header- and tail-size, which are the same for all blocks in the pool:

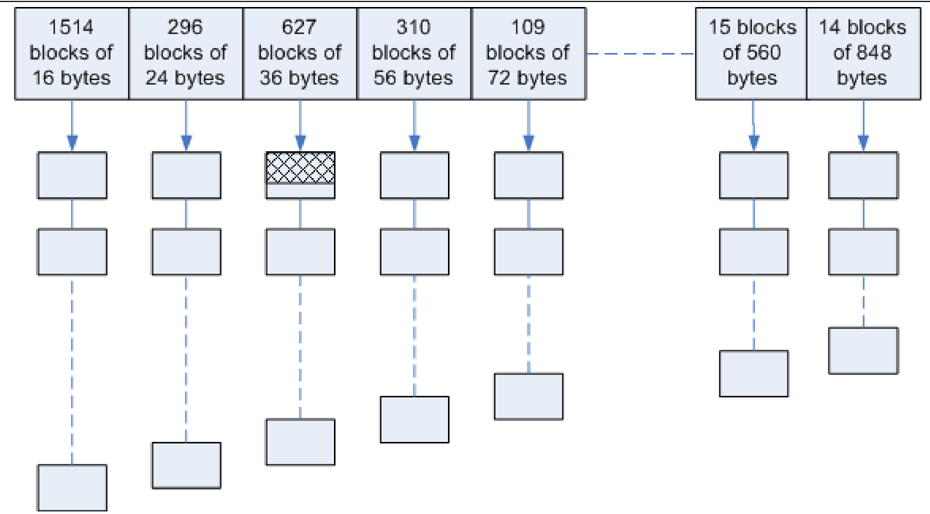
$$upaddress = lowaddress + headersize + size + tailsize - 1$$

Since both the size, lower- and upper-address of a block will be needed in subsequent chapters this slightly redundant definition of the set of blocks will be assumed.

$$BL_{i,j} = \langle lowaddress, size, upaddress \rangle$$

In general, bins with larger blocks will have fewer blocks. Figure 2.2 shows an example of a pool configuration.

Figure 2.2 An example of a pool configuration. The hatched area indicates the effective section of a block of 36 bytes after a request for 25 bytes.



An allocation[1] of a block in the pool can be described by a 5-tuple containing the start- and end-time of the allocation, the id's of the processes that made the allocation and de-allocation,

[1]Note that here an allocation is defined as the occupation of a fraction of memory for a period of time, as opposed to the *event* where memory is allocated.

and finally the size (in bytes) needed by the process.

$$BA_{i,j} = \langle begintime, endtime, alloc_proc, dealloc_proc, size \rangle$$

In the following, the bins in $\{Bin_i\}$ will be considered separately, hence the index is dropped for the sake of brevity.
Individual blocks in the pool are fully allocated or not allocated at all. It is not possible to allocate only part of a block in the pool. Consequently, the amount of memory granted by the memory allocator might differ from the amount requested by the process, leading to a certain amount of what will be referred to as *memory waste* or *internal memory fragmentation* ([Ran69]). Figure 2.2 depicts the allocation of a block of 36 bytes as a response to a request for 25 bytes. The hatched area represents the part of the block actually used by the process. Conversely, the non-hatched area of the same block represents its wasted memory.
The reason the allocator maintains this pool structure is to limit the impact of *external memory fragmentation* ([Ran69]). External memory fragmentation occurs when allocating and de-allocating blocks of different sizes. In dynamic structures, a de-allocation creates a hole in the contiguous memory space. When subsequent allocations do not require the exact amount of memory supplied by this hole, it is only partially reused. As the remaining unused space in such a hole becomes small, it becomes unusable. This phenomenon occurs repeatedly and the small unusable holes accumulate into large amounts of un-allocatable memory. Naturally, processes with large lifetimes suffer more from external memory fragmentation than short-lived processes do.
By defining a predefined structure, the pool limits the impact of external memory fragmentation as no unused block will ever become (close to being) un-allocatable. External fragmentation of the pool can however lead to decreased speed of some allocators. Furthermore, as mentioned before, this predefined structure introduces internal memory fragmentation. Another disadvantage of the pool is that it can only allocate up to moderately-sized blocks. Requests for bigger chunks of memory require a different allocation approach.

Heap

In contrast to the pool, the heap does not have a predefined structure.
The necessity for the heap arises from the fact that the pool can only deal with allocations up to a certain size. On the downside, it generally suffers from external fragmentation as allocations of variable sizes are made in its dynamic structure. By checking whether an allocation can be made in the pool before considering the heap, the variation in allocation size in the heap is decreased. Consequently, the level of external heap-fragmentation is decreased.
The heap has a lower- and upper address, a total size $|Heap|$ and a header- and tail-size, equal for all allocations made therein. Allocations in the heap can have an arbitrary size as long as there is still sufficient unallocated memory left. Therefore, internal fragmentation is absent in the heap structure.
An allocation in the heap is a 6-tuple containing the start- and end-time of the allocation, the lower-address the id's of the allocating and de-allocating process and the size (in bytes) of the allocation:

$$HA = \langle begintime, endtime, address, alloc_proc, dealloc_proc, size \rangle$$

The upper-address of an allocation in the heap follows implicitly from the lower-address, size, header- and tail-size of the heap:

$$upaddress = address + headersize + size + tailsize - 1$$

2.1.2 Log Format

The data generated by the mobile device is supplied through a (textual) log containing information about memory allocation and de-allocation in a specified format. A separate header file

contains the information about the memory layout. On behalf of the heap, it includes the size, header- and tail-size and upper- and lower-address. On behalf of the pool it includes the header- and tail-size and the bin-block layout.
The log file is divided into lines, each containing exactly one allocation or de-allocation event[2] into the pool or heap. A few sample log lines are:

```
<0001h 26m 32s 059 316> MCU OS Task: 2 @ 0x013a69e6,0x013a6ce2 Block alloc
ptr 0x000ae494 size 34 set  3

<0001h 26m 32s 059 391> MCU OS Task: 2 @ 0x013a7578,0x013a7608 Block dealloc
ptr 0x000ae494 set 3

<0001h 26m 32s 555 304> MCU OS Task: 30 @ 0x013a1cba,0x01537e36 Heap dealloc
ptr 0x030c2b8c

<0001h 26m 32s 554 810> MCU OS Task: 30 @ 0x013a1532,0x01537a22 Heap alloc
ptr 0x030c2b8c size 912
```

Each line also describes some information related to the allocation or de-allocation event. These are:

- The time of the allocation or de-allocation, given by a number of hours, minutes, seconds, milliseconds and microseconds
- The id of the process that made the allocation or de-allocation
- Information on the function that called the allocation or de-allocation
- Whether it concerns an allocation or de-allocation into the heap, or one of the bins
- Whether it concerns an allocation or de-allocation
- The memory-address of the allocation or de-allocation
- The size in bytes needed by the process (this only occurs on allocation lines and is not always equal to the size of memory granted by the pool)
- The bin the allocation occurs in (this only appears when the allocation is made in the pool)

Information such as ”MCU OS” and ”no wait” are operating system specific tokens and can be ignored for the purpose of this thesis.
The de-allocation of a piece of allocated memory is not always present in the log. The profiler could for example have been preempted before all de-allocations were made, or the log could have been shortened by discarding a trailing section. This is a shortcoming of the log and it could lead to a potentially misleading analysis as part of the behavior of the allocator is ignored. Another possibility however, is that these allocation events were actually not de-allocated by the allocator and that this behavior points to memory leaks.

2.2 Software Configuration Management Systems

This section presents the second example of time-dependent dynamic software artifacts. The data model presented is based on the infrastructure for data mining of different kinds of software repositories presented in [VT].
Software Configuration Management (SCM) is defined in [Pre01] as a “set of activities designed

[2]Note that the log specifies allocations through separate allocation and de-allocation events.

to control change by identifying the work products that are likely to change, establishing relationships among them, defining mechanisms for managing different versions of these work products, controlling the changes imposed, and auditing and reporting on the changes made". Large software projects often utilize a system for supporting the SCM methodology. CVS (Concurrent Versions System[3]) and Subversion[4] are examples of such systems. They keep track of all work and changes in the set of files related to the software project, hereby facilitating collaboration between several developers. These management systems are mainly used to store source code, which is widely recognized as the "main asset of the software engineering economy" ([Str00]). This makes these systems an excellent up-to-date source of information for the analysis of projects following a corrective approach ([VT]). SCM systems hence facilitate the analysis of the evolution and productivity of potentially large software projects.
Typical questions facing software project managers, targeted by the methods in this thesis include:

- How is project-wide activity distributed?
- Which files are heavily modified and by whom?
- Which groups of files are developed together?
- How are these related files distributed over the folder structure?
- At what moments did a mayor release of the project occur?

2.2.1 Evolution Data Model

This section describes the data model for the software evolution data. It globally follows the model presented in [VT], with some modifications. The evolution data is made available through the CVSgrab tool ([VT06a]). This tool generates a log by performing a depth-first traversal of the SCM system's repository root. This yields a log which consists of a listing of file entries in alphabetical order. Each file entry is followed by an enumeration of the branches it belongs to and finally a list of commit entries. These commit entries determine the version of a file in a specific branch. For the purpose of this thesis, code branching is ignored, i.e. only the "main" branch, or trunk, of the repository is assessed. Other branches could be analyzed separately. However, these branches are often short-lived and only span a handful of files.
The following presents a model for the evolution data extracted from these logs. The central element of an SCM system is a *repository* R which stores the evolution of a set of NF files:

$$R = \{F_i \mid i \in 1 \dots NF\}$$

In a repository, each file F_i is stored as a set of NV_i versions:

$$F_i = \{V_{i,j} \mid j \in 1 \dots NV_i\}$$

Each version is a tuple with several *attributes*. The most typical ones are: The name of the author who committed it, the commit time, a log message, the number of lines added and the number of lines removed with respect to the previous version:

$$V_{i,j} = \langle author, committime, comment, \#linesadded, \#linesremoved \rangle$$

The *author* and *comment* attributes are unstructured categorical attributes. *committime*, while strictly seen a discrete attribute, is considered as a continuous one. Naturally, the two modification attributes, *#linesadded* and *#linesremoved* are of integer type. They are acquired through a *diff*-like tool.

[3]http://www.nongnu.org/cvs/
[4]http://subversion.tigris.org/

2.3 Generic Data Model

This thesis focuses on the analysis of time-dependent software artifacts in general. The two data models described in sections 2.1.1 and 2.2.1 are too detailed to fit this purpose. This section introduces the notion of an *element* as the generic basis for discussing the proposed visualization methods for time-dependent software artifacts. It also discusses a restriction on the type of dataset that can be subdivided into elements and consequently the type of dataset that can be visualized using those methods.

Time-dependent software artifacts consist of a set of attributed elements $\mathcal{E} = \{e^i\}$, for $i = 1 \ldots |\mathcal{E}|$. An element e^i is the Cartesian product of a time-interval in the range $[T_{min} \ldots T_{max}]$ and an interval in range $[L_{min} \ldots L_{max}]$ of an ordered set L, the main parameter of analysis. Hence, it consists of a start- and an end-time and a start- and an end-offset in L:

$$e^i = \langle s^i_T, e^i_T, s^i_L, e^i_L \rangle \tag{2.1}$$

The following property enforces a restriction on the set L. All pairs of elements with intersecting time intervals have disjoint intervals in set L, or stated formally:

$$\big(\forall e^i : e^i \in \mathcal{E} : \big(\forall e^j : e^j \in \mathcal{E} \wedge e^i \neq e^j : Overlap_T(e^i, e^j) \Rightarrow \neg Overlap_L(e^i, e^j)\big)\big) \tag{2.2}$$

where $Overlap_T : \mathcal{E} \times \mathcal{E} \rightarrow Bool$ and $Overlap_L : \mathcal{E} \times \mathcal{E} \rightarrow Bool$ indicate whether two elements' time intervals and offset intervals overlap respectively, i.e.:

$$Overlap_T(e^i, e^j) = \left(s^i_T < e^j_T\right) \wedge \left(s^j_T < e^i_T\right) \tag{2.3}$$

$$Overlap_L(e^i, e^j) = \left(s^i_L < e^j_L\right) \wedge \left(s^j_L < e^i_L\right) \tag{2.4}$$

The visualization methods proposed in chapter 3, specifically the layout model used, will explicitly use this non-overlapping property. Additionally, each element has several attributes of categorical, continuous or integer data.

The notion of an *element*, described above, establishes a basis for the generic discussion of the visualization methods in chapter 3. Chapter 4 will show how the data models for the two applications translate to this generic model. Strictly seen, any data model having main analysis parameter L that satisfies property 2.2 is adequate for visual analysis using the methods presented in this thesis. Naturally, not all data models will benefit equally from these methods as the usefulness of any visualization relies heavily on the underlying semantics of the data. By tweaking the parameters of the visualization techniques that follow in chapter 3 however, a high level of correlation between visual entities and their corresponding high-level structures can be reached for many different types of data models.

Chapter 3

Visualization Methods

This chapter describes methods for getting insight into the behavior of time-dependent software processes, such as the memory allocator and the repository management system introduced in chapter 2.
The main proposal is to produce this insight by means of interactive visualizations of the time-dependent data. By visualizing abstract data, this data is transformed into a form in which it is better understood by the human brain, thus reducing the cognitive load of the brain ([Swe86]). Furthermore techniques that make use of color, lighting and texturing hold a great deal of potential for enhancing the visualization as they appeal directly to the brain.
This chapter consists of several sections each building on top of its predecessor. Section 3.1 starts by describing the basic visualization model. The remainder of the chapter refines the model defined in section 3.1. Section 3.2 explains how applying cushion-like textures to the rectangles makes them more distinguishable. Section 3.3 elaborates on some methods for mapping the rectangles to the limited space on the screen without parting with too much information. Section 3.4 presents a metric bar that displays statistical information on the elements active at a specific point in time. Section 3.5 introduces the notion of "clustering" where several chunks of closely-related rectangles are grouped together to form a more global view of the data. Additionally, each section lists the advantages and disadvantages of the technique discussed.

3.1 Visualization Model

In general a visualization model for information visualization is a function from an abstract dataset to a representation on the screen. A more detailed description of the model for the dataset at hand is a function

$$VM : \mathcal{E} \rightarrow (Position, Size, Shape, Color, Texture, Lighting) \tag{3.1}$$

In visualization practice, this function consists of several algorithmic steps: data importing, data filtering, data mapping and rendering. These constitute what is called the *visualization pipeline* ([CMS99]). This thesis concentrates on the latter two elements of the visualization pipeline, namely data mapping and rendering. Data mapping is concerned with the layout of the data on the screen. The layout portion assigns geometric position, dimension and shape to a non-visual element. This layout can be modeled as a function LM:

$$LM : \mathcal{E} \rightarrow (Position, Size, Shape) \tag{3.2}$$

which fixes the layout-related portion of the model. Similarly, function

$$RM : \mathcal{E} \rightarrow (Color, Texture, Lighting) \tag{3.3}$$

describes the rendering-related portion of the model. These sub-models are described in sections 3.1.1 and 3.1.2 respectively.
For a visualization function to be effective, it should satisfy a few important properties. A visualization function should be:

1. *Efficiently computable*: Naturally an uncomputable or very inefficient visualization function is of very limited use.
2. *Visually invertible*: The mapping of attributes of interest to visual elements should be one to one, for the brain to be able to grasp the meaning of the representation.
3. *Easily invertible*: Values used for for example color, shading and position should be as different as possible for the user to be able to extract information from the visualization easily.

The visualization model presented in this thesis is a mapping from the data model to a 2-dimensional orthogonal plot with time on the x-axis and offset in parameter L, introduced in section 2.3, on the y-axis. Several visual attributes such as color, shading and texture can be manipulated to convey additional information, as will become apparent in the next few sections.

3.1.1 Layout Model

Property 2.2 ensures that no two elements have overlapping intervals for L at any given point in time. Consequently, plotting time against L and mapping it to a visual counterpart induces a pairwise-disjoint 2-dimensional subdivision. This planar rectangular subdivision is the general layout of the visualization.
The rectangular representation of an element e^i, projected to the screen is the Cartesian product of two rational-valued intervals, of which values in viewport $[0 \ldots X_{max}] \times [0 \ldots Y_{max}]$ are rendered. In the following these rectangular on-screen representations of elements, will be referred to as segments. The layout portion of a segment, corresponding to element e^i, is represented by its upper and lower values for both axes:

$$s^i = \langle s^i_x, e^i_x, s^i_y, e^i_y \rangle$$

Let $\mathcal{S}$ be the set of segments:

$$\mathcal{S} = \{s^i \mid i = 1 \ldots |\mathcal{E}|\}$$

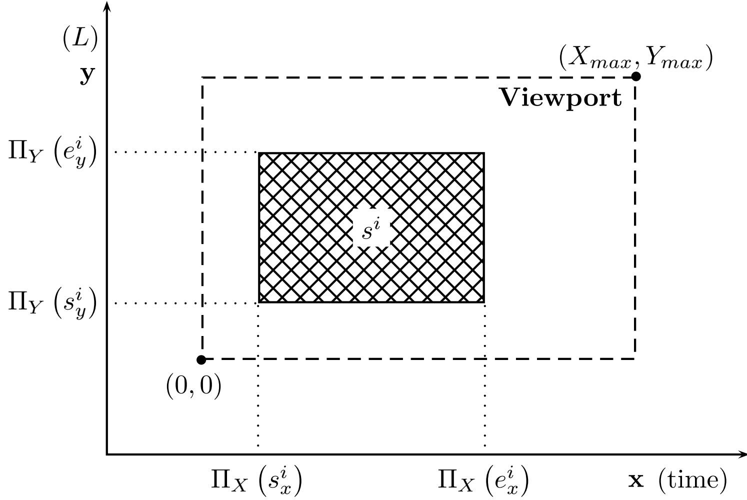
The mapping from element e^i to s^i occurs by applying linear functions $\Pi_X : T \rightarrow \mathbb{R}$ and $\Pi_Y : L \rightarrow \mathbb{R}$ to values s^i_T, e^i_T and s^i_L, e^i_L respectively:

$$\Pi_X(t) = \sigma_X \frac{X_{max}}{T_{max} - T_{min}}(t - T_{min}) + \delta_X \quad (3.4)$$

$$\Pi_Y(l) = \sigma_Y \frac{Y_{max}}{L_{max} - L_{min}}(l - L_{min}) + \delta_Y \quad (3.5)$$

Here, σ_X, σ_Y, δ_X and δ_Y are user-supplied parameters for zooming in to specific areas of the rectangular representation.
The additional attributes of element e^i do not affect the layout.

Figure 3.1 2-dimensional visualization layout of a segment



This approach offers several advantages:

- It is compact. A lot of elements, typically hundreds of thousands, can be simultaneously shown on the screen
- No screen space is wasted. Empty areas convey actual information, for example they indicate memory fragmentation for the memory allocator data.
- It is simple and efficiently computable. The coordinates of the rectangle representing an element are merely the result of a linear function applied to attributes of the element.
- It is well-organized. As a consequence of property 2.2 the visualization has no overlapping segments. Furthermore all segments are perfect rectangles and are aligned with the axes.
- It is intuitive. Segments on the screen represent elements directly. Furthermore, the horizontal axis is traditionally used to represent time, which is also done here.
- It is effective. As the mapping function is an injective one, the information on the screen can be translated back to the data-set. The "visually invertible"-property is hereby satisfied.

2D or 3D

An interesting and ever recurring discussion is on whether to use a 2-dimensional or a 3-dimensional visualization. Several researchers have promoted 3D software visualizations for time-dependent data ([TLTC05],[RCM93]). Despite their appeal however, 3D visualizations introduce a number of difficulties compared to 2D ones. These include, but are not limited to ([KG05, CM01, CM02, CM04]):

- Object occlusion. Depending on the viewpoint, one object might be blocking the viewer's view of another.
- Awkward navigation. Navigation through the 3D world using tools operating in two dimensions (mice, trackballs) is not easily accomplished in a straightforward way. It is usually done by holding down a key-, or mouse-button to change the orientation of the navigating plane, and moving the mouse to navigate within this plane. This makes 3D navigation harder to get acquainted with, compared to the simple pan and zoom of 2D navigation.

- Awkward selection. As a result of object occlusion and awkward navigation, the task of selecting an object becomes a tedious one.
- Obscured relations. Through perspective projection, relations between the sizes of objects are obscured. As the object is further away from the viewer it is drawn smaller than an object of the same size located close to the viewer.
- Slow rendering. The extra dimension creates a higher degree of calculations to be performed. Expensive operations like anti-aliasing applied to large data sets may very well cause the rendering of the visualization to become unacceptably slow.

For the purpose of this thesis, these disadvantages outweigh the benefit of the extra dimension, hence a 2D visualization is used.

3.1.2 Rendering Model

Rendering describes the process of assigning color-, lighting- and texture attributes to an already laid out data object. In this case, the laid out data object is a 2-dimensional axis-aligned rectangle, also called a segment.
Additional element-attributes can for example be encoded into the color of its segment. Mapping more than one attribute onto a coloring scheme is unintuitive and is not easily invertible. Other rendering-related parameters, like texture and lighting could be used for representing a second or even third attribute as is done in [VT06b]. However, as segments become very small (width or height a few pixels or less), there is not always enough room for these parameters to be represented effectively. For this reason, at most one attribute is mapped to rendering-related parameters at a time. Furthermore, all methods presented in the remainder of this thesis require only coloring and texturing. Consequently, these two suffice as rendering parameters and lighting is dropped. The rendering portion of a segment s^i is hence a 2-tuple containing a color and a texture item:

$$s^i = \langle col^i, tex^i \rangle$$

The discussion on the rendering model continues with an exploration of the different mappings of segment attributes to colors. Different attributes can be explored by switching between coloring schemes. This basic rendering model will exhibit some shortcomings as will become apparent in the next few sections. Despite its limited effectiveness when applied to small segments, texturing can be used effectively to limit some of these shortcomings by refining the rendering model. Sections 3.2 and 3.5.5 will discuss some applications of texturing that fit this purpose.

Color maps: In mapping data attributes to colors, one can distinguish between two types of attributes:

- *Continuous*: Continuous data attributes are quantified attributes like time or size where large values may represent favorable or unfavorable scenarios.
- *Categorical*: Categorical data attributes are attributes for identifying or grouping an entity. Comparisons other than equality between different categorical attributes do not make sense.

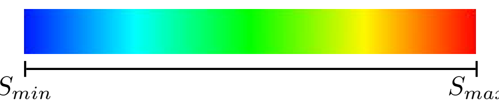
The purpose of coloring a continuous data attribute of a segment is to give an indication of the relative value for that segment. Take a general color map $CM : [0 \ldots \mathcal{D}] \rightarrow color$ which maps a scalar in the given range to a color. For an attribute with range $[S_{min} \ldots S_{max}]$ adapted color map function $\mathcal{C}$ maps scalar s to a color using color map CM:

$$\mathcal{C}(s) = CM\left(\mathcal{D}\frac{s - S_{min}}{S_{max} - S_{min}}\right) \tag{3.6}$$

In the following, any reference to a color map will refer to this adapted form of color map.
The most common color map for continuous attributes, the rainbow color map (figure 3.2), is a

hue-based linear scale from blue, through a rainbow of colors, to red. Here, blue traditionally represents favorable and red unfavorable values.

Figure 3.2 Adapted color map $\mathcal{C}$ based on the rainbow color map

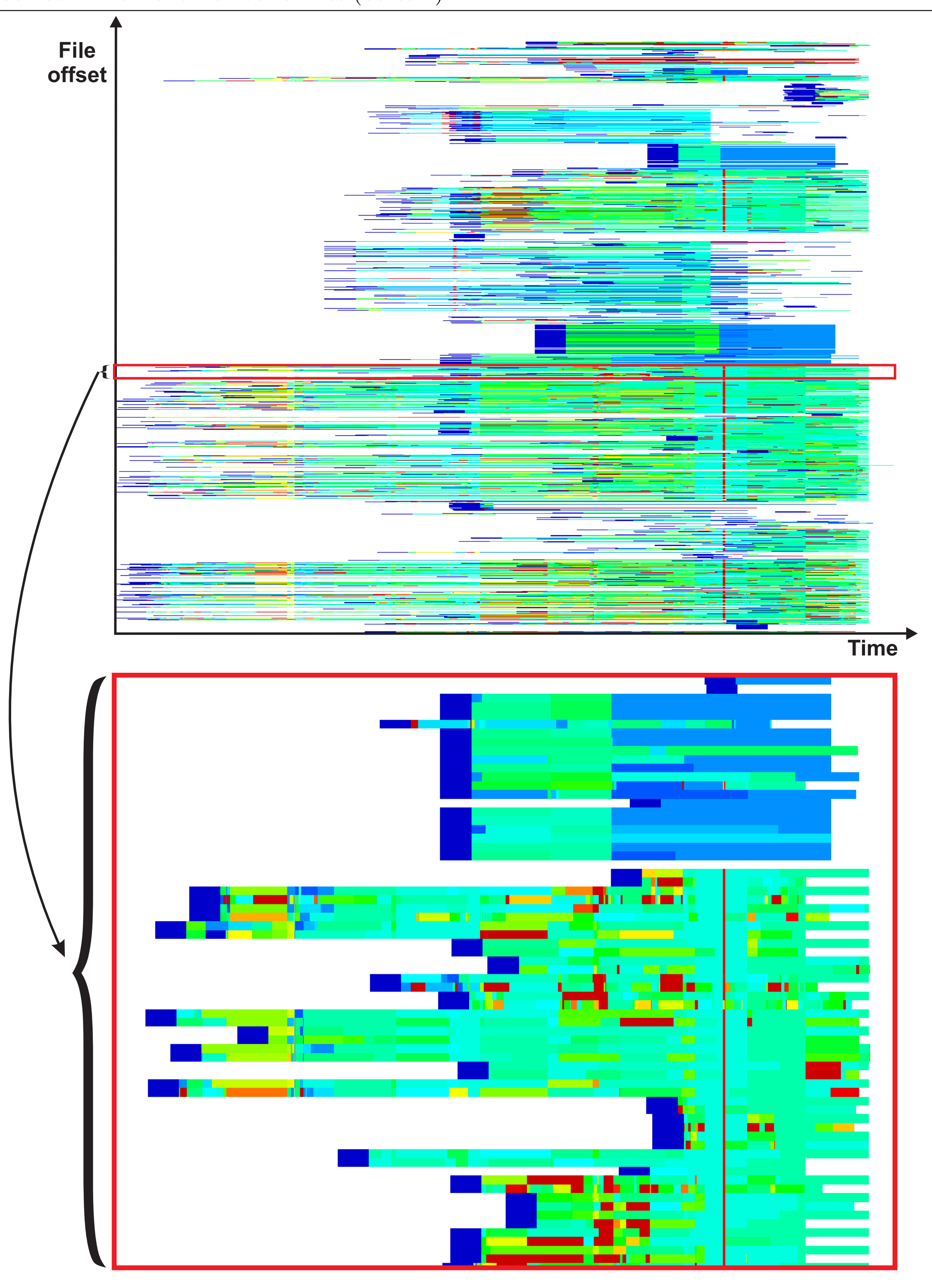


Perceptually however, this scale is far from linear as is immediately apparent from the figure. Equal steps in the scale do not correspond to equal steps in color. As a result, this color map can lead the user to infer structure which is not present in the data and to miss details that lie completely within a single color region ([BRT95], [RLK92], [RT93]). Some other types of well-known color maps include black-and-white color maps, blue-white-red color maps and heat color maps. Furthermore, a number of tools exist for creating custom color maps (ColorBrewer[1], [BRT95]). Despite its pitfalls, the rainbow color map is chosen as the main continuous color map in this thesis, due to its intuitive mapping to good, neutral and bad values, its familiarity and its general acceptance amongst the researching community.
For categorical data attributes the purpose of coloring is to distinguish between different discrete values. Ideally this is done by a discrete color map specifying a different color for each occurring value. Unfortunately, the human eye can only distinguish between 6 to 10 colors easily. A color map that cycles through around 10 highly different colors only provides a partial solution to this problem, as potentially many different values are mapped to the same color. Section 3.5.5 discusses how textures can diminish a specific instance of this problem.
Together with the layout model of section 3.1.1 this basic rendering model maps a set of elements to a (potentially large) collection of disjoint rectangles on the screen. Figure 3.3 shows an example application of this preliminary visualization model[2]. The following sections describe techniques that enhance this model.

[1] http://www.colorbrewer.org/

[2] Some of the figures in this chapter and the next show sections of the visualization of the memory allocator or SCM system examples. The exact details for these examples will follow in chapter 4.

Figure 3.3 A 2-dimensional dense visualization of the files in a software project, using rectangles for different versions, colored by level of change using the rainbow color map (top) and a vertically zoomed in view of a number of files (bottom)



3.2 Cushioning

The first shortcoming of the preliminary rendering model of section 3.1.2 is revealed when two adjacent segments have the same color. Figure 3.3 (bottom) shows that for these neighboring segments of equal color, it is not very apparent where one ends and the other begins. Drawing borders around the segments clutters the overall image when the density of the segments is high. A more elegant solution is obtained by applying a parabolic function to the color-intensity of the segment, creating a 3D cushion-like surface, first introduced in ([vWvdW99]). This approach is also used in EZEL [VTvW04], a visualization tool from a similar application domain. EZEL assesses the efficiency of a peer to peer file sharing network and offers a similar layout where time is mapped against an offset in a file.
Cushioning is however only effective when the segments are at least a few pixels long and wide.

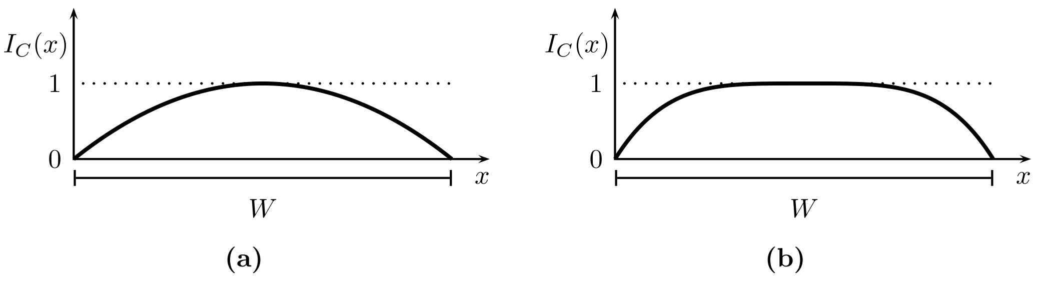
3.2.1 Parabolic cushions

A simple way of creating a cushion is by applying a texture to the segments, using a parabolic function in x- and y direction, translated into an intensity map. The two components are combined by multiplication. Let $I_C(x, y) \rightarrow [0 \ldots 1]$ denote the intensity of cushion $C = W \times H$ for $x \in [0 \ldots W]$ and $y \in [0 \ldots H]$, where W and H represent the width and the height of the cushion, respectively.

$$I_C(x, y) = \alpha * (1 - \left(\frac{| 2x - W |}{W}\right)^\gamma) * (1 - \left(\frac{| 2y - H |}{H}\right)^\gamma) \quad (3.7)$$

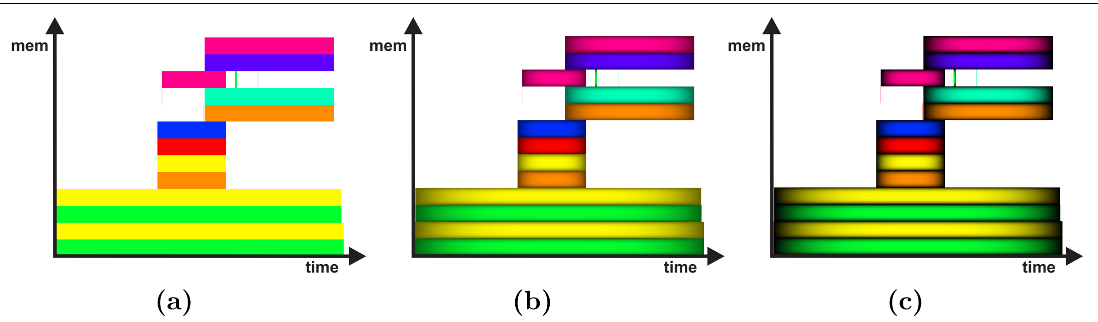
where $\alpha \in [0 \ldots 1]$ and $\gamma \in \mathbb{R}^+$ are constants for altering the intensity and the steepness of the cushion, respectively. Figure 3.4 sketches the profile of this cushion in one dimension for $\gamma = 2$ and $\gamma = 4$. Notice the difference in steepness at the extremities. The α parameter does not affect the profile, but rather the intensity with which the cushion is applied.

Figure 3.4 Profile of a parabolic cushion for (a) $\gamma = 2$ and (b) $\gamma = 4$



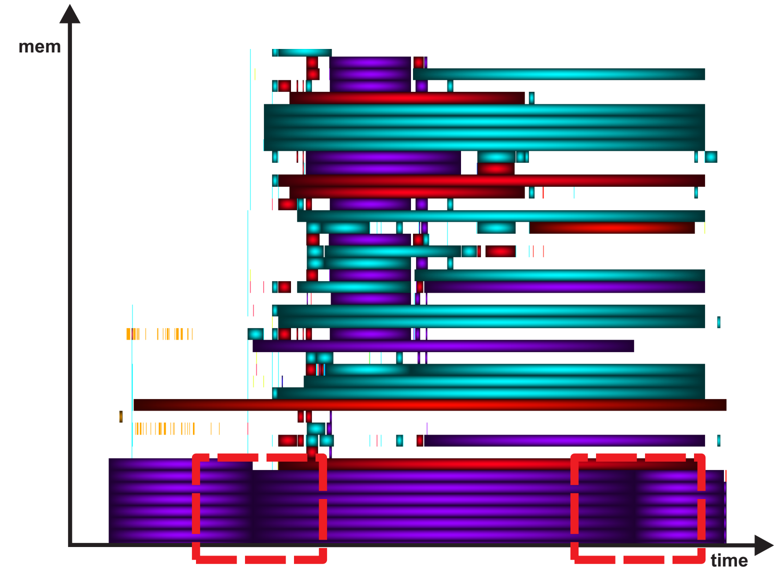
Although cushions are often desired for visual segregation between segments, they can sometimes disturb the greater picture similarly to bordered segments. For this reason the user can adjust the visual strength of the cushion using the α value. Figures 3.5 (a), (b) and (c) show collections of these cushions as they are represented on the screen, with $\alpha = 0$, $\alpha = 0.6$ and $\alpha = 1$ respectively.
The parabolic cushioning approach is very simple and effective. A disadvantage of this type of cushion is that the texture scales with the surface, leading to large dark areas when visualizing a group of long segments (See figure 3.6). This side-effect of parabolic cushions implicates a higher level structure of the underlying data, a visual artifact which is exploited in section 3.5.5, when the focus lies on amplifying high-level structure. For localized analysis however, these darkened areas merely blur the visual segregation between long segments, especially when colored using the same dark color.

Figure 3.5 Parabolic cushions with $\gamma = 4$ and (a) $\alpha = 0$, $\alpha = 0.6$ and (c) $\alpha = 1$



(a) (b) (c)

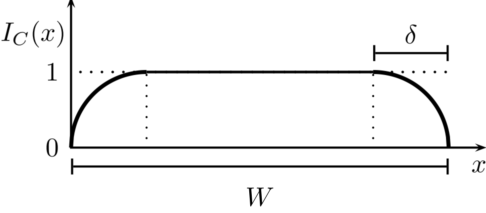
Figure 3.6 Side effect of parabolic cushions in visualization of memory allocator



3.2.2 Plateau cushions

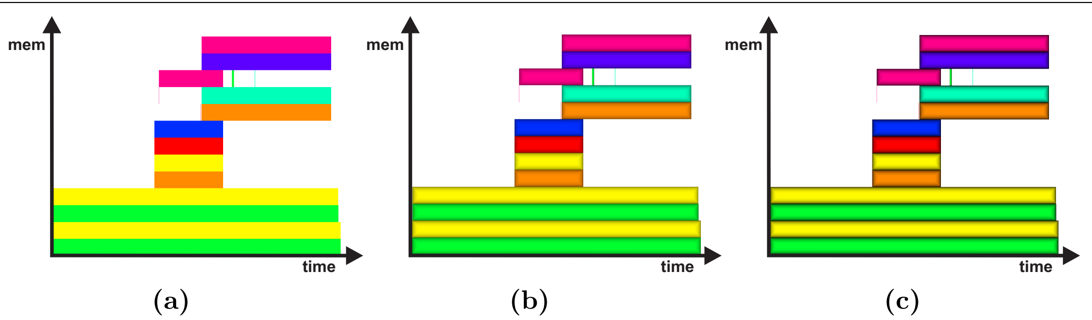
A second approach cushions a surface through three different intensity functions for both dimension sketched in figure 3.7.

Figure 3.7 Profile of a plateau cushion



This yields eight distinct intensity maps and one constant intensity middle section as shown in figure 3.8. Parameter δ denotes the width of the slopes at the cushion's extremities. This

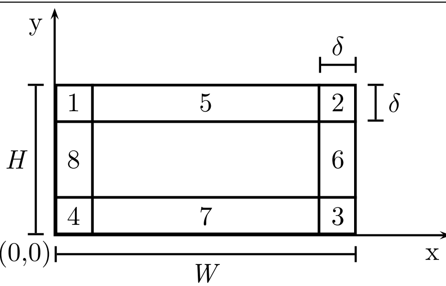
Figure 3.9 Plateau cushions with (a) $\alpha = 0$, $\alpha = 0.6$ and (c) $\alpha = 1$



(a) (b) (c)

type of cushion, first introduced in [LNVT05], is referred to as a plateau cushion because of the constant intensity of the middle section.

Figure 3.8 Subdivision of a plateau cushion



The basis for the intensity functions of sections 1 through 8 is a single function $\mathcal{I}$, applied to either dimension:

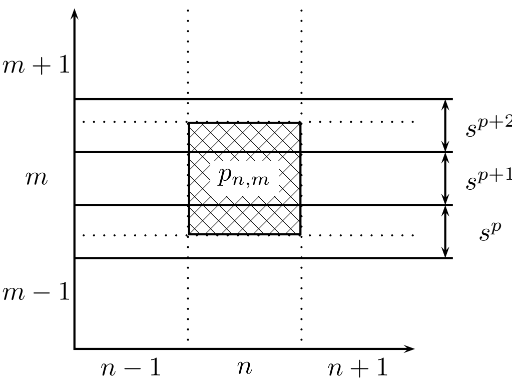
$$\mathcal{I}(x) = (x/\delta - 1)^2 \tag{3.8}$$

They are given by functions 3.9 to 3.16 respectively. The center of the cushion has maximal intensity across its surface.

$$I_{C_1}(x, y) = \alpha * max\,(\mathcal{I}(x), \mathcal{I}(H - y)) \tag{3.9}$$
$$I_{C_2}(x, y) = \alpha * max\,(\mathcal{I}(W - x), \mathcal{I}(H - y)) \tag{3.10}$$
$$I_{C_3}(x, y) = \alpha * max\,(\mathcal{I}(W - x), \mathcal{I}(H - y)) \tag{3.11}$$
$$I_{C_4}(x, y) = \alpha * max\,(\mathcal{I}(x), \mathcal{I}(y)) \tag{3.12}$$
$$I_{C_5}(x, y) = \alpha * \mathcal{I}(H - y) \tag{3.13}$$
$$I_{C_6}(x, y) = \alpha * \mathcal{I}(W - x) \tag{3.14}$$
$$I_{C_7}(x, y) = \alpha * \mathcal{I}(y) \tag{3.15}$$
$$I_{C_8}(x, y) = \alpha * \mathcal{I}(x) \tag{3.16}$$

Figures 3.9 (a), (b) and (c) show collections of these cushions as they are represented on the screen, with $\alpha = 0$, $\alpha = 0.6$ and $\alpha = 1$ respectively. When these cushions are scaled, only the non textured area of the cushion is scaled, eliminating the darkened areas seen before. The disadvantage of this approach is that these cushions, when drawn using hardware accelerated texture mapping, are less efficient to compute, as eight different textures need to be mapped to different parts of the segment instead of only one. Furthermore segments that are smaller than 2δ in either direction still require scaling of the textures in order to make them fit. The techniques presented in section 3.3 eliminate the former disadvantage.

Figure 3.10 Segments s^p, s^{p+1} and s^{p+2} (partly) covering pixel $p_{n,m}$



3.3 Sub-sampling

An artifact of the discrete nature of graphical displays is that many small segments, or parts of a segment could cover one pixel. Naturally one pixel can only exhibit one color. What color should the pixel be given?

The situation is depicted in figure 3.10. It is not simply a hypothetical situation. It occurs quite frequently in practice, for example when the number of blocks or number of files exceeds the number of pixels in the memory allocator, respectively the SCM system application. Say pixel p is covered by n segments $s^1, \ldots, s^n \in \mathcal{S}$, which show categorical or continuous attributes $a^1, \ldots, a^n$ and $col(p)$ indicates the color of pixel p. The color of the background of the visualization is given by col_B. Furthermore, let f_p^i denote the fraction of p's total area covered by a segment s^i, and f_p^B the remaining uncovered fraction of that pixel. As a corollary to property 2.2 the following holds:

$$\sum_{i=1}^{n} f_p^i \leq 1 \tag{3.17}$$

$$f_p^B = 1 - \sum_{i=1}^{n} f_p^i \tag{3.18}$$

The base color of a pixel refers to the color supplied by the current color map, ignoring the contribution of the segments' textures. The following describes how the base color of a pixel can be acquired.

Recall from section 3.1.2 that for segment s^i, col^i is either derived from categorical or continuous attribute a^i using a color map CM. Sub-sampling is considered as a method for combining the attributes $a^1, \ldots, a^n$ representatively. It can be applied at two different stages of the rendering pipeline. In figure 3.11, functions SS_1 and SS_2 represent the sub-sampling functions to be specified later in this section.

Figure 3.11 Portion of the rendering pipeline where sub-sampling is applied to segments $s^1, \ldots, s^n$ covering pixel p (partially), (a) after and (b) before the color of the segment is determined

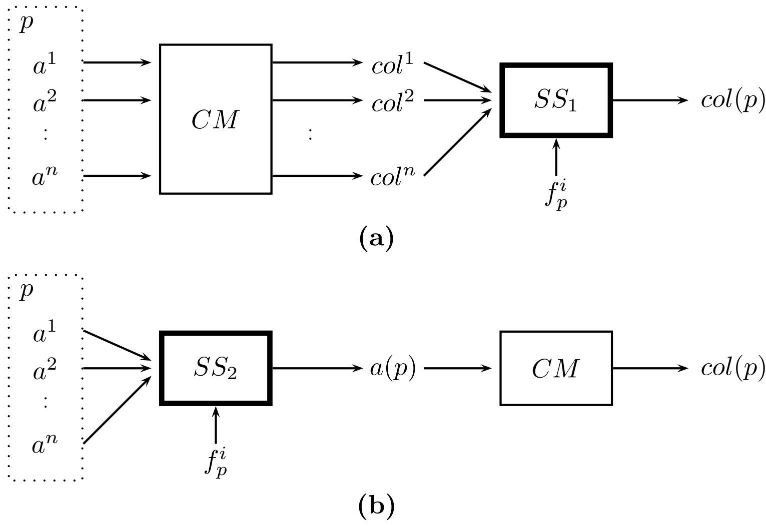


Figure 3.11 (a) shows sub-sampling applied after the color of a segment is chosen using the current color map. The colors of the segments covering a pixel p are combined into the color of the pixel directly by function SS_1, using the set of coverage fractions $\{f_p^i\}$. In the remainder of this thesis this type of sub-sampling will be referred to as *color sub-sampling*. In figure (b) sub-sampling (SS_2) is first applied to attributes a^i, for $i \in 1, \ldots, n$, using the coverage fractions, and the resulting value is then mapped to the p's color using the current color map. This type of sub-sampling will be referred to as *scalar sub-sampling* and can only be meaningfully applied to attributes for which addition and multiplication are defined.

The difference between these strategies is subtle, but important to dwell on[3]. Color sub-sampling, applied to a continuous attribute using the rainbow color map for example, potentially yields a color which is not in the color map. Scalar sub-sampling offers a better solution for continuous attributes. As the scalars $s^1, \ldots, s^n$ are first combined into a single scalar $at\,(p)$ for pixel p and only then translated into a color, the result gives an adequate collective representation of the concerning segments.

Categorical attributes complicate things further. Scalar sub-sampling between two categorical values a and b yields a scalar in the range $[a \ldots b]$, which could translate to a color which is highly different from colors $CM(a)$ and $CM(b)$. In this case color sub-sampling provides better results as the user is potentially able to reason on the composition of the displayed color. The resulting color might however still belong to a different categorical value. This ambiguity is a limitation of both the scalar and the color sub-sampling approach and in some cases makes visual inversion of rendered sub-sampled data impossible for categorical attributes. Despite this shortcoming, sub-sampling remains useful as it does enable the user to identify *that* something is presented by some pixel, although it might not always be clear *what* exactly is presented. Areas of interested can subsequently be zoomed into for more information on the details of that area. Figure 3.12 illustrates an example of the scenarios sketched above.

[3]Literature research however, yielded no previous work in which a similar distinction is made.

Figure 3.12 Scenario of different types of results possible when combining color or scalar sub-sampling with categorical or continuous attributes. Notice how color sub-sampling of continuous data leads to a color not in the color map and how scalar sub-sampling of categorical attributes leads to an unrelated color.

	Categorical	Continuous
Color map	2 → red; 3 → blue; 4 → green	(color gradient)
Color sub-sampling	4 / 2 → olive	1 / 0 → purple
Scalar sub-sampling	4 / 2 → blue	1 / 0 → green

It seems that in general color sub-sampling works best for categorical attributes, while scalar sub-sampling does so for continuous attributes. Using this point of view, the sub-sampling function is now either a function $SS_1 : color^n \rightarrow color$ or $SS_2 : \mathbb{R}^n \rightarrow \mathbb{R}$. These functions are describes in sections 3.3.1 and 3.3.2, respectively.

3.3.1 Color Sub-sampling

Function SS_1, applied to the colors of segments $s^1, \ldots, s^n$ together with coverage fractions $f_p^1, \ldots, f_p^n$ generating $col(p)$, can be described by several types of functions:

- **Replace**: Pixel p could be given a color picked arbitrarily from $s^1, \ldots, s^n$. OpenGL's glRect*() command colors a pixel with the current color, if that pixel is covered by the specified rectangle by more than some minimal fraction MF. A sequence of these commands results in a $col(p)$ that equals the color of the last drawn rectangle that covered it by more than MF. The order of the sequence of segments drawn is more or less arbitrary, hence this results in the specified relation. Figure 3.13 (a) shows the results of this approach. Since OpenGL rendering can be done in hardware, this solution's advantage is that it is the fastest of the bunch.

- **Maximum**: p could be given the color of the segment that covers it the most, so

$$col(p) = col(s^m) \tag{3.19}$$

  with $s^m \in \{s^1, \ldots, s^n\}$ and f_p^m maximal. In this way, small segments (with width or height $<$ size of 1 pixel) are likely to not be shown at all due to them being overclouded by a larger neighboring segment. If small rectangles are of little interest to the user because of their limited size, this is an acceptable approach.

- **Linear**: The most fair solution is to combine the colors of a segment according to their coverage ratio:

$$col(p) = \sum_{i=1}^{n} f_p^i * col^i + f_p^B * col_B \tag{3.20}$$

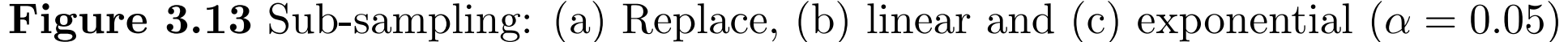
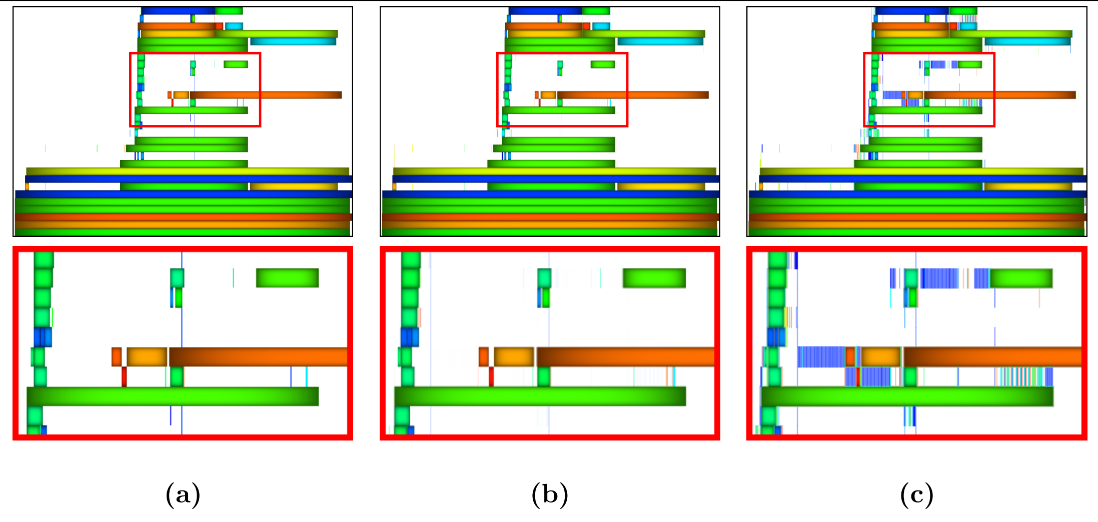


Figure 3.13 Sub-sampling: (a) Replace, (b) linear and (c) exponential ($\alpha = 0.05$)



(a) **(b)** **(c)**

This is shown in figure 3.13 (b). The background component contributes to the color of the pixel if and only if less than holds in Corollary 3.17.

- **Importance-based**: A more flexible solution blends the colors of the segments, biased according to their coverage fraction, using an exponential function. An example of such a function is:

$$col(p) = \frac{\sum_{i=1}^{n}(f_p^i)^\alpha * col(s_i) + f_p^B * col_B}{\sum_{i=1}^{n}(f_p^i)^\alpha + f_p^B} \tag{3.21}$$

where parameter α is a parameter indicating the bias-level [4]. The denominator normalizes the result using the the total exponentiated contributions. For $\alpha < 1$, segments with a small area are "biased" to become more apparent in the final image. The lower the value of α, the more evenly the segments $s^1, \ldots, s^n$ contribute to the color of alpha. Furthermore, isolated thin segments become more clearly visible as the background component remains constant. To improve things further, segments expanding over at least 1 entire pixel can be left out of the equation as they are already "clearly" visible. The effect is shown in figure 3.13 (c) for $\alpha = 0.05$. Conversely, for $1 < \alpha$, small segments are filtered out. This is useful, for example, for uncluttering the image, when thin segments can safely be ignored. By choosing $\alpha = 1$ this function describes the exact same relation as the linear function above. By making α a user-specified parameter, thin segments can be emphasized, or filtered out based on what the user considers important. This technique is introduced here, and is suitably called *importance-based sub-sampling*.

The *maximum* approach offers a more sensible and predictable solution compared to the *replace* approach. The largest segment covering a pixel would namely be the most visible if the discussed limitation of graphical displays was absent. The *replace* approach might still favor a smaller segment for the simple reason that it was drawn after the bigger one. Both approaches draw at most a single segment per pixel, which amounts to a regular undersampling of the dataset. More gravely, segments of subpixel width, smaller than MF become invisible with the *replace* approach. As mentioned before however, it is much faster as it can be carried out in hardware. The *linear* approach in turn gives a more fair result than the *maximum* one. As discussed above however, by mixing different colors together, new colors are created which may confuse the user

[4] Note that the meaning of the symbol α is overloaded and has no relation to the cushion-strength parameter of section 3.2.

as these may not always be easily visually invertible. Importance-based sub-sampling is more flexible as it is capable of favoring larger or smaller segments according to the user's perception of what is important. By favoring small elements for example, it could emphasize elements which would otherwise be almost invisible. However, this approach disturbs the one-to-one mapping of segments to their positions on the screen which can be misleading to the unaware user.

3.3.2 Scalar Sub-sampling

Function SS_2 combines a range of continuous attributes $a^1, \ldots, a^n$ relating to segments $s^1, \ldots, s^n$, using coverage fractions f_p^i, into a combined attribute for pixel $a(p)$. Subsequently, $a(p)$ is mapped to a color using color map CM and then combined with col_B into $col(p)$, similarly to functions discussed in section 3.3.1. This can be summarized by the following functions:

- **Linear**: SS_2 combines $a^1, \ldots, a^n$ into $a(p)$ by:

$$a(p) = \sum_{i=1}^{n} f_p^i * at(s^i) \tag{3.22}$$

  The background component does not contribute to $a(p)$, as it has no senseful value for this attribute. Next $col(p)$ is determined by:

$$col(p) = \left(\sum_{i=1}^{n} f_p^i \right) CM(a(p)) + f_p^B * col_B \tag{3.23}$$

  This function blends the background color with the color yielded by color mapped attribute $a(p)$.

- **Importance-based**: Similarly to color sub-sampling, an importance-based method can also be applied for combining attributes $a^1, \ldots, a^n$. An exponential is used in both steps:

$$a(p) = \frac{\sum_{i=1}^{n} (f_p^i)^\alpha * at(s^i)}{\sum_{i=1}^{n} (f_p^i)^\alpha} \tag{3.24}$$

  $col(p)$ is then determined by:

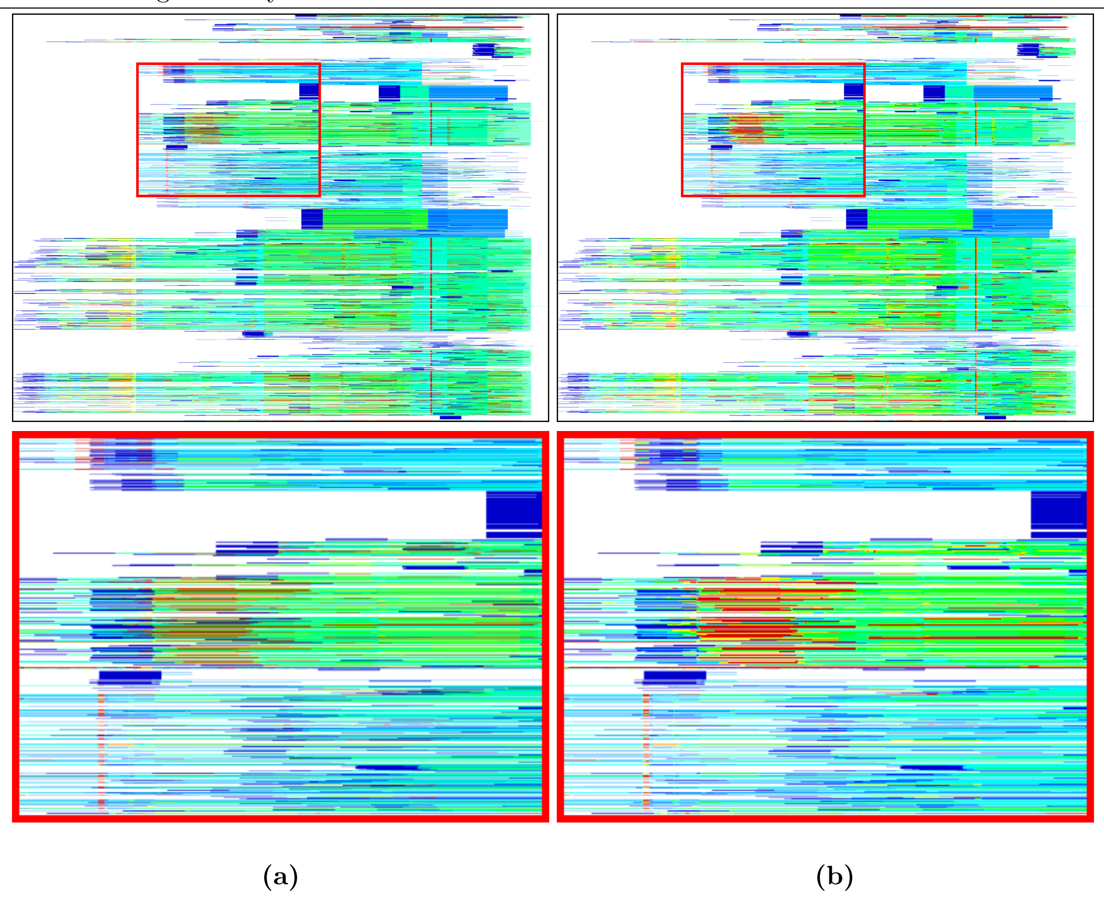
$$col(p) = \frac{\left(\sum_{i=1}^{n} (f_p^i)^\alpha\right) CM(a(p)) + f_p^B * col_B}{\sum_{i=1}^{n} (f_p^i)^\alpha + f_p^B} \tag{3.25}$$

The benefit of scalar sub-sampling over color sub-sampling, becomes apparent in densely populated areas containing segments colored by continues attributes. This difference can be seen in figure 3.14.

3.4 Metric Bar

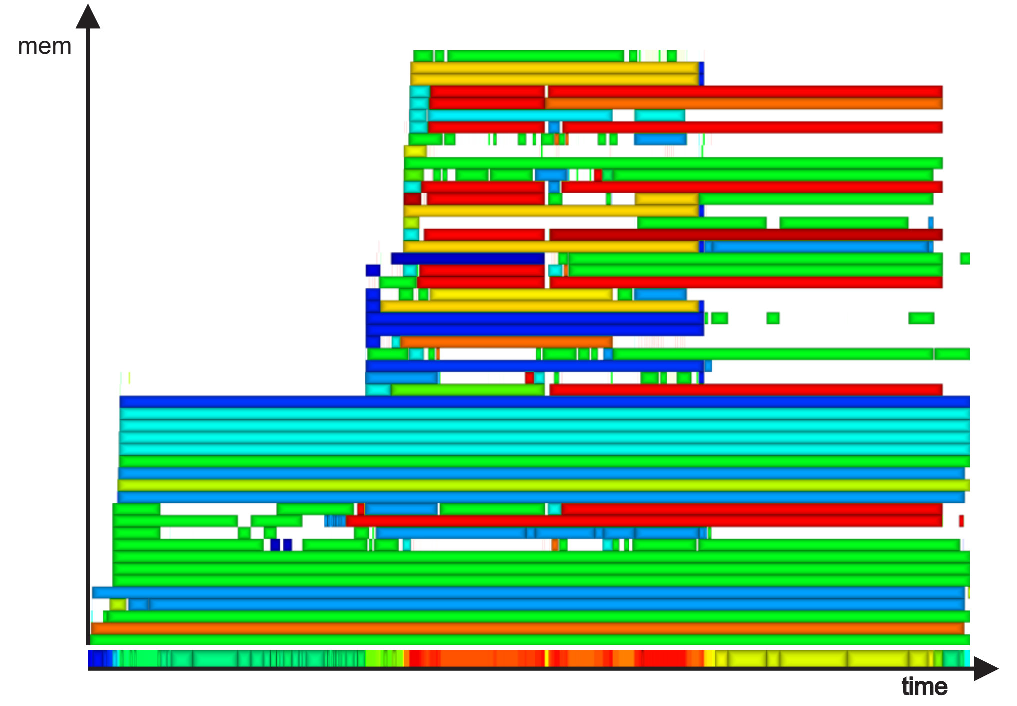
An adjacent 1-dimensional horizontal bar extends the visualization by visualizing specific metrics for the visualized period of time. The values for these metrics are normalized to utilize the complete range of the color map. The metric bar is especially useful for quickly finding moments in time that are of particular interest. This technique is also applied in [VT]. The cushioning and sub-sampling techniques described above are also applied to the metric bar. In this manner, long periods of inactivity can be detected unambiguously. This in contrast to [VT], where no explicit visual hint about the segregation between separate eventless periods is available.
The choice of which metric to display is task-specific and will be discussed in chapter 4. A similar approach is possible in vertical direction, displaying metrical information on elements sharing

Figure 3.14 The main branch of an SCM system repository colored by level of change, using color sub-sampling (a) and scalar sub-sampling (b). Notice the difference in color, especially around the high-activity outlined area.



the same value for parameter L. An example metric bar, displaying a continuous accumulated attribute for the visualization above it using the rainbow color map, is given in figure 3.15.

Figure 3.15 Visualization of a bin in memory for the dynamic memory allocator application, colored by fragmentation. The metric bar shown below the main visualization depicts the cumulative occupancy.

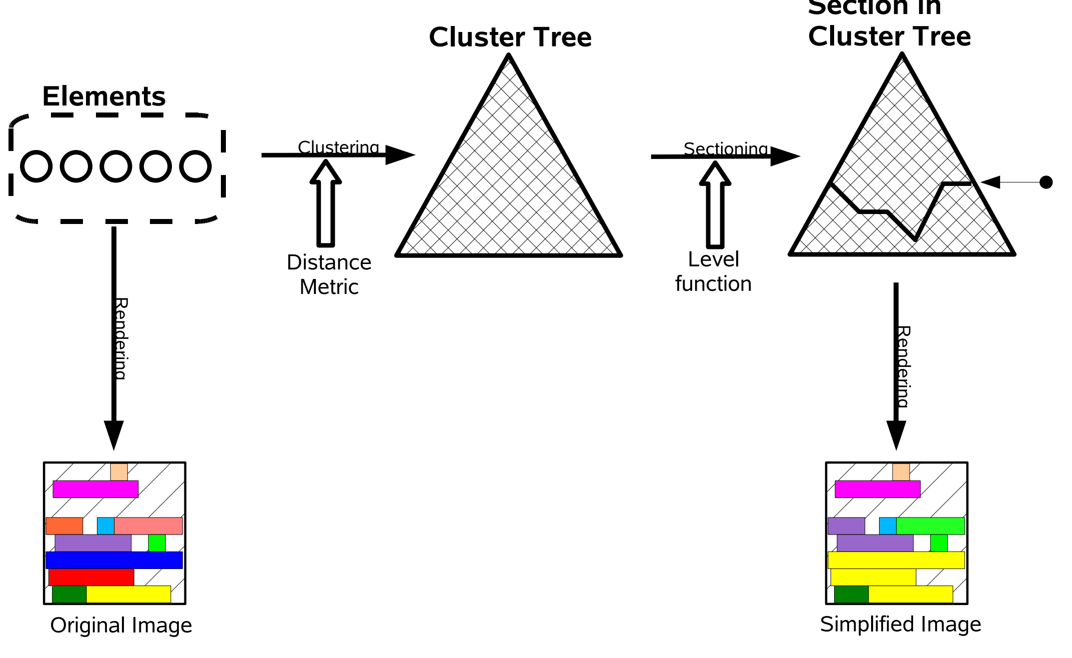


3.5 Hierarchical Agglomerative Clustering

In its current form, the visualization is capable of accommodating the user in forming a decent image of the local evolution of the data set. However, some global or quasi-global questions remain that the visualization can not yet answer adequately.
Hierarchical agglomerative clustering algorithms provide a multi-level partitioning of data. These algorithms work bottom-up, and iteratively merge the two closest clusters with respect to some distance metric, until a certain criterion is satisfied. Applying such an algorithm to the set of elements $\mathcal{E}$, yields a weighted tree structure, also referred to as a dendrogram, with the elements as leaves. Each section in this tree represents a partitioning of the elements which can be rendered to the screen. The user can then move this section up and down the tree interactively, until a desired level of structure is reached. This dynamic partitioning of the data serves as a visual aid for supporting the user in detecting the artifact's high-level structure.
The general steps involved in this process are summarized in the pipeline of figure 3.16. First, the set of elements are input to the hierarchical agglomerative clustering algorithm, based on a distance metric, yielding the clustering tree. Next, a user-selected level function and threshold determine the subset of clusters which are finally rendered to the screen.
Section 3.5.1 presents a number of different distance metrics, while section 3.5.2 describes the details of the agglomerative clustering algorithm. Section 3.5.3 details the rendered subset selection process, also referred to as *sectioning*. In figure 3.16, color is used to indicate the cluster a segment belongs to. Section 3.5.4 discusses some problems with this approach, while

section 3.5.5 introduces *interleaved cushioning*, a new technique that alleviates these problems through the use of shaded cushions.

Figure 3.16 Global pipeline of the clustering algorithm



3.5.1 Distance Metric

The agglomerative clustering algorithm depends on a distance metric for measuring the level of similarity between items in the dataset $\mathcal{E}$. A distance metric is a function $d : \mathcal{X}^2 \rightarrow \mathbb{R}$ satisfying the following properties for all $x, y, z \in \mathcal{X}$,

- $d(x, y) > 0$, if $x \neq y$
- $d(x, y) = 0$, if x = y
- $d(x, y) = d(y, x)$
- $d(x, z) \leq d(x, y) + d(y, z)$

The choice of this distance metric for the clustering algorithm highly determines the type of pattern that will be visible. This section describes a number of different distance metrics over elements $e^p, e^q \in \mathcal{E}$.

Vertically-adjacent fair distance metric

$$d_1(e^p, e^q) = \begin{cases} |s^p_T - s^q_T| + |e^p_T - e^q_T|, & \text{if } neighbors(e^p, e^q) \\ \infty, & \text{if } \neg neighbors(e^p, e^q) \end{cases} \tag{3.26}$$

where $neighbors : \mathcal{E}^2 \rightarrow Bool$ is defined as follows:

$$neighbors(e^p, e^q) = ((e^p_L \equiv s^q_L) \vee (s^p_L \equiv e^q_L)) \wedge (s^p_T \leq e^q_T) \wedge (s^q_T \leq e^p_T) \tag{3.27}$$

This metric is depicted in figure 3.17 (a) for 2 y-adjacent elements. It is adequate for finding compact structures of any size. Of course, structures of greater size will be better visible. Since

the function does not take the size of the elements into consideration and patterns containing small elements are generally harder to interpret than bigger ones, the user will likely prefer smaller elements to be clustered first. A more adequate metric would therefore cluster smaller elements before the bigger ones. To this end, the size of the elements needs to be added to the equation.

Vertically-adjacent biased distance metric

Consider the following distance metric, again over $e^p, e^q \in \mathcal{E}$:

$$d_2(e^p, e^q) = \frac{A(e^p) + A(e^q)}{2 * A_{max}} * d_1(e^p, e^q) \tag{3.28}$$

where for $e^i \in \mathcal{E}$, $A(e^i)$ is the area of that element:

$$A(e^i) = (e^p_T - s^p_T) * (e^p_L - s^p_L) \tag{3.29}$$

A_{max} is the area of the largest possible area for any element. This can without loss of generality be taken as $(T_{max} - T_{min}) * (L_{max} - L_{min})$. It is used to normalize the area of the elements to a value in the range $[0, 1]$. This metric is depicted in figure 3.17 (b) for two y-adjacent elements. It is adequate for finding similar structures, determining their compactness and finding correlations between these structures and attributes. It is however not adequate for finding vertically segregated structures, since it requires clustered elements to be vertically adjacent.

Vertically-independent fair distance metric

For some datasets, set L is not ordered in a strict way. These datasets are such that all elements have the same interval size. Consequently, their corresponding segments form a sequence of fixed height horizontal stripes in the visualization. The vertical ordering of these stripes may be arbitrary, or quasi-arbitrary. The distance metrics presented above however, impose a very strict limitation on the type of composition of clusters, through the neighboring requirement. Only elements adjacent in vertical direction can be clustered. Although desired for datasets with strict orderings in set L, a more flexible one is needed for the formerly presented type of dataset. By dropping this strict neighboring requirement clustering also becomes useful for the datasets not ordered in L. This yields the following cost function over $e^p, e^q \in \mathcal{E}$:

$$d_3(e^p, e^q) = (|s^p_T - s^q_T| + |e^p_T - e^q_T|) \tag{3.30}$$

Furthermore, this type of distance metric also potentially reveals hidden patterns for datasets strictly ordered in L. The main drawback of this distance metric, compared to the adjacent distance metric is that by dropping the vertical restraint the number of possible clusterings increases significantly. For building the complete clustering tree, generally $O(|\mathcal{E}|^2 \; log \; |\mathcal{E}|)$ comparisons need to be made. Additionally, visual segregation becomes harder as cluster representations may span several segments not belonging to it.

Vertically-independent biased distance metric

Finally, note that a combination of distance metrics d_2 and d_3 is also possible. This yields distance metric d_4, which also biases smaller elements:

$$d_4(e^p, e^q) = \frac{A(e^p) + A(e^q)}{2 * A_{max}} * d_3(e^p, e^q) \tag{3.31}$$

This metric is depicted in figure 3.17 (c) for two vertically segregated elements.

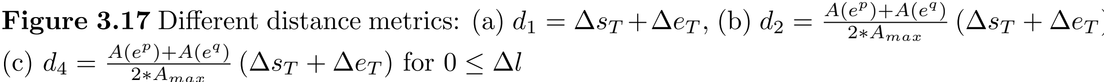
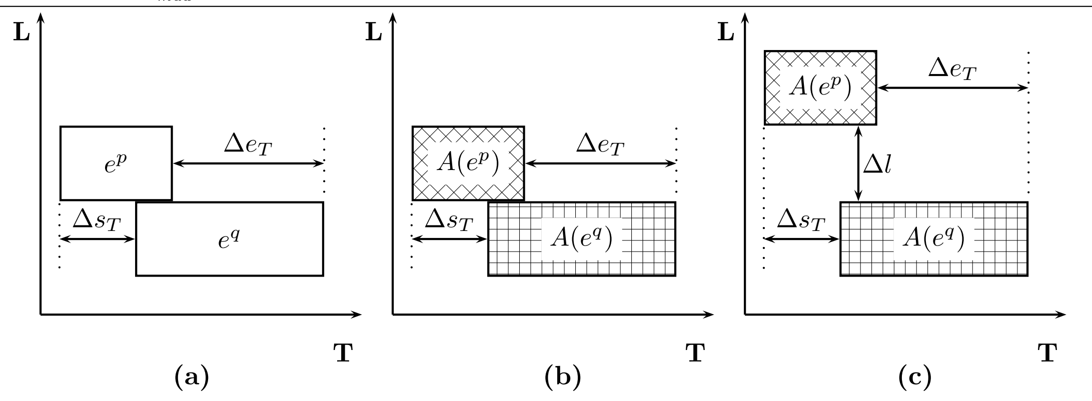


Figure 3.17 Different distance metrics: (a) $d_1 = \Delta s_T + \Delta e_T$, (b) $d_2 = \frac{A(e^p)+A(e^q)}{2*A_{max}}(\Delta s_T + \Delta e_T)$ (c) $d_4 = \frac{A(e^p)+A(e^q)}{2*A_{max}}(\Delta s_T + \Delta e_T)$ for $0 \leq \Delta l$



3.5.2 Clustering Model

This section describes an agglomerative, bottom-up algorithm for the construction of a clustering tree, using the distance metrics described above, which induces a multi-level partitioning, or hierarchy among elements. This hierarchy is primarily achieved by grouping several highly correlated elements together in a recursive manner. These groups, referred to as clusters, are sets of smaller clusters. Initially, each cluster contains a single element. Each element occurs as a singleton in exactly one of these initial clusters, also referred to as the *base set*. Let $\mathcal{CL}_1$ be the base set containing the $|\mathcal{E}|$ elements. In each iteration of the agglomeration algorithm, recursive clustering set $\mathcal{CL}_{i+1}$ is constructed from set $\mathcal{CL}_i$ by picking two clusters $c^p, c^q \in \mathcal{CL}_i$ and replacing them by a single cluster $(c^p \cup c^p)$, or stated formally:

$$\mathcal{CL}_{i+1} = (\mathcal{CL}_i \setminus \{c^p, c^q\}) \cup \{c^p \cup c^q\} \tag{3.32}$$

for $i = 1, \ldots, n-1$, where $\mathcal{CL}_n$ is the first set in which all pairs of clusters have infinite distance. The clustering tree can be constructed intermingled with the sequence of sets $\{\mathcal{CL}_i\}$. The procedure for the construction of this tree is given in algorithm 1. Actually, for some restrictive

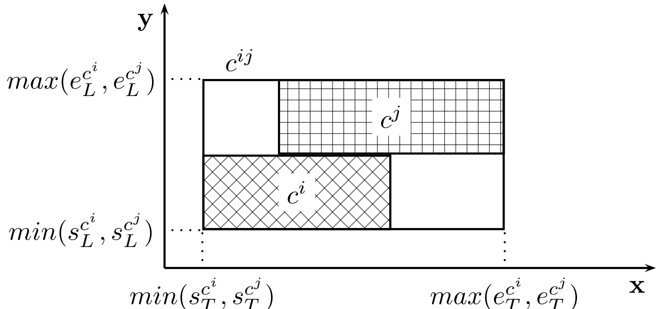
Algorithm 1 Build the clustering tree

Require: Set of initial clusters $\mathcal{CL}$ and distance metric d
Ensure: V_F and E_F contain the nodes, respectively the edges of the clustering tree F

1: $V_F \leftarrow \mathcal{CL}$;
2: $E_F \leftarrow \emptyset$;
3: **while** $(\exists a, b \in \mathcal{CL} : d(a,b) \neq \infty)$ **do**
4: Select $a, b \in \mathcal{CL}$ such that $d(a,b)$ is minimal;
5: $c \leftarrow \{a \cup b\}$;
6: $V_F \leftarrow V_F \cup c$;
7: $E_F \leftarrow E_F \cup \{c, a\} \cup \{c, b\}$;
8: $\mathcal{CL} \leftarrow \setminus\{a, b\}$;
9: **for all** $x \in \mathcal{CL}$ **do**
10: Recompute distances between x and c;
11: **end for**
12: $\mathcal{CL} \leftarrow \mathcal{CL} \cup c$;
13: **end while**

distance metrics, this algorithm creates a clustering *forest*, instead of a single tree or dendrogram,

Figure 3.18 Outline of a cluster c^{ij}, composed of clusters c^i and c^j



as the distance between some pairs of elements is infinite. Furthermore, notice how in each step $|\mathcal{CL}|$ decreases by one, due to the merging of exactly two clusters. Consequently, the *while*-loop is executed $|\mathcal{CL}| - |\mathcal{FR}|$ times, where $|\mathcal{FR}|$ is the number of trees in the resulting forest.
Literature proposes two types of linkage, for the aggregation of the two closest clusters, based on the distance metric defined on the base set. Each have a different scheme for the computation of distances between clusters.

- *Single-linkage aggregation*: The distance between two clusters is determined by the two closest elements in their base sets.
- *Complete-linkage aggregation*: The distance between two clusters is determined by the two furthest elements in their base sets.

In this thesis, for reasons which will become apparent in section 3.5.5, the distance between two clusters is defined in an alternative way. Instead of basing the distance between two clusters directly on the elements in their base sets, the elements in the base set of the clusters iteratively determine the outline of the cluster. Similar to elements, a cluster c^i has layout attributes $s_T^{c^i}$, $e_T^{c^i}$, $s_L^{c^i}$ and $e_L^{c^i}$. For cluster c^i, these are recursively defined as:

$$s_T^{c^i} = \begin{cases} min(s_T^{c^j}, s_T^{c^k}), & \text{if } c^i = \{c^j, c^k\} \\ s_T^l, & \text{if } c^i = \{e^l\} \end{cases}$$
$$e_T^{c^i} = \begin{cases} max(e_T^{c^j}, e_T^{c^k}), & \text{if } c^i = \{c^j, c^k\} \\ e_T^l, & \text{if } c^i = \{e^l\} \end{cases}$$
$$s_L^{c^i} = \begin{cases} min(s_L^{c^j}, s_L^{c^k}), & \text{if } c^i = \{c^j, c^k\} \\ s_L^l, & \text{if } c^i = \{e^l\} \end{cases}$$
$$e_L^{c^i} = \begin{cases} max(e_L^{c^j}, e_L^{c^k}), & \text{if } c^i = \{c^j, c^k\} \\ e_y^l, & \text{if } c^i = \{e^l\} \end{cases}$$

This outline scheme is sketched in figure 3.18. In addition, for each cluster c^i, function A describes its area:

$$A(c^i) = \begin{cases} A(c^j) + A(c^k), & \text{if } c^i = \{c^j, c^k\} \\ A(e^l), & \text{if } c^i = \{e^l\} \end{cases} \tag{3.33}$$

Now the alternative definition of distance between two clusters c^p and c^q can be described by function d^c in terms of the distance metrics, d_1, d_2, d_3 and d_4, for elements discussed in section 3.5.1. As clusters have similar layout attributes as elements, the domain of these metric functions

can be broadened to include clusters. Now d_i^c is given by:

$$d_i^c(c^p, c^q) = \begin{cases} max(d_i^c(c^j, c^k), d_i^c(c^l, c^m), d_i(c^p, c^q)), & \text{if } c^p = \{c^j, c^k\} \wedge c^q = \{c^l, c^m\} \\ max(d_i^c(c^j, c^k), d_i(c^p, e^q)), & \text{if } c^p = \{c^j, c^k\} \wedge c^q = \{e^q\} \\ max(d_i^c(c^l, c^m), d_i(c^q, e^p)), & \text{if } c^p = \{e^p\} \wedge c^q = \{c^l, c^m\} \\ d_i(e^p, e^q), & \text{if } c^p = \{e^p\} \wedge c^q = \{e^q\} \end{cases} \quad (3.34)$$

The maximum is taken over the child node weights to ensure that all nodes on the paths from the leaves to the root of the clustering tree are ascending in weight. This property ensures that d^c is again a distance metric. These distances can be iteratively computed and stored in the nodes, during the construction of the tree. Unlike single-linkage or complete-linkage aggregation, this definition of distance between clusters introduces a limited, but accumulating error level as clustering advances. The typical weakness of the data model, however, makes this limited error level acceptable, and structural patterns in the data still discoverable.
For restricted distance metrics d_1 and d_2, an optimization of algorithm 1 is possible, and highly fruitful. Steps 9 and 10 of this algorithm recompute the distances between all pairs in recursive cluster set $\mathcal{CL}$. For distance metrics d_1 and d_2 however, only a small subset of these distances are finite. By additionally keeping a neighborhood graph, using a set of weighted edges $\mathcal{N}$ between the clusters in set $\mathcal{CL}$, only finite distances between two clusters are recalculated after this neighborhood graph has been built. The optimized version of algorithm 1, for restricted distance metrics, is shown in algorithm 2. *Update* is given in algorithm 3. Figure 3.19, sketches

Algorithm 2 Build the clustering tree for restrictive distance metrics

Require: Set of initial clusters $\mathcal{CL}$ and distance metric d_i
Ensure: V_F and E_F contain the nodes, respectively the edges of the clustering tree F

1: $V_F \leftarrow \mathcal{CL}$;
2: $E_F \leftarrow \emptyset$;
3: $\mathcal{N} \leftarrow \{c^p, c^q\} \mid c^p = \{e^i\} \wedge c^q = \{e^j\} \wedge d_i(e^i, e^j) \neq \infty$;
4: **while** $(\exists a, b \in \mathcal{CL} : d_i^c(a, b) \neq \infty)$ **do**
5: Select $a, b \in \mathcal{CL}$ such that $\{a, b\} \in \mathcal{N} \wedge d_i^c(a, b)$ is minimal;
6: $c \leftarrow \{a \cup b\}$;
7: $V_F \leftarrow V_F \cup c$;
8: $E_F \leftarrow E_F \cup \{c, a\} \cup \{c, b\}$;
9: $\mathcal{CL} \leftarrow \mathcal{CL} \setminus \{a, b\}$;
10: $Update(N, a, b)$;
11: $\mathcal{CL} \leftarrow \mathcal{CL} \cup c$;
12: **end while**

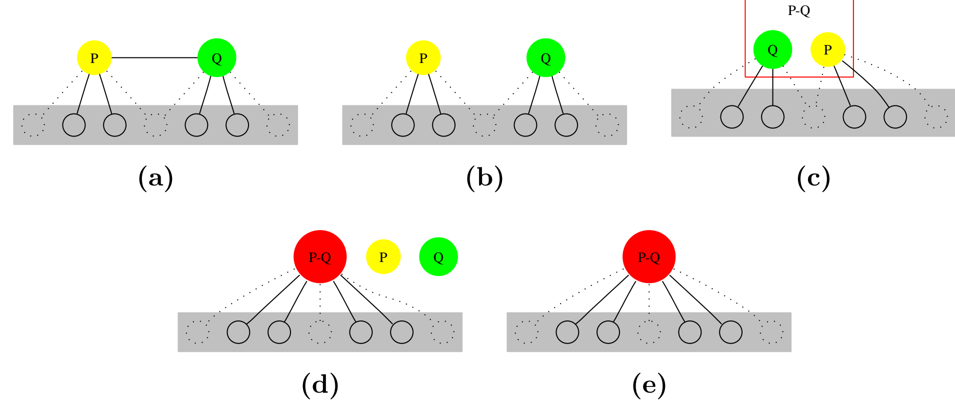
Algorithm 3 Update the neighborhood graph

Require: $\mathcal{N}$ is a set of edges containing edge $\{a, b\}$
Ensure: Edge $\{a, b\}$ has been appropriately replaced in $\mathcal{N}$

1: $\mathcal{N} \leftarrow \mathcal{N} \setminus \{a, b\}$;
2: **for all** $\{v, v'\} \in \mathcal{N} \wedge ((v = a) \vee (v = b))$ **do**
3: $\mathcal{N} \leftarrow \mathcal{N} \setminus \{v, v'\}$;
4: $\mathcal{N} \leftarrow \mathcal{N} \cup \{a \cup b, v'\}$;
5: Recompute distance between $\{a \cup b\}$ and v';
6: **end for**

the merging of two clusters, as preformed by algorithm 2. Note that each isolated component in the initial neighborhood graph yields a tree in the clustering forest.

Figure 3.19 Merging clusters ”P” and ”Q” into composite node ”P-Q”, and the effect on the neighborhood graph: (a) Before clustering, (b) edge $\{P, Q\}$ is removed, (c) P and Q are clustered, (d) the neighborhoods are updated and (e) individual clusters P and Q are removed



Phased Tree Construction

The inherently quadratic nature of the clustering tree produced by the vertically-independent distance metrics introduces a difficulty with the implementation of the algorithm presented above. Step 4 of algorithm 1 involves finding the pair of minimum distance and clustering these into a new node. Depending on $|\mathcal{E}|$, this step is executed a large number of times. Ideally, a sequential data structure $\mathcal{D}$, containing the pairs of elements is kept sorted on distance, so $\mathcal{D} = \{p_1, \ldots, p_n\}$, where $d(p_1) \leq d(p_2) \leq \cdots \leq d(p_n)$. Here, p_i is a commutative pair of clusters $\left(c^{p_i^1}, c^{p_i^2}\right)$ and $d(p_i)$ denotes the distance between these two clusters. In each iteration of the outer loop in algorithm 1, a pair of minimal distance is then extracted from $\mathcal{D}$. For the vertically-independent distance metrics, $\mathcal{D}$ initially contains $\frac{1}{2}|\mathcal{E}|(|\mathcal{E}| - 1)$ pairs, since all pairs of elements have finite distance. Typically, $|\mathcal{E}|$ is in the order of hundreds of thousands. Very optimistically estimating the amount of memory taken to represent a single pair at 2 bytes, the creation of the clustering tree for 100000 elements already takes around 10 gigabytes of main memory. Naturally, even larger datasets, which are not uncommon would take even more. Most of today’s personal computers cannot cope with such large datasets without the use of a much slower pagefile. The other extreme is that no edges are kept in memory. In every iteration of the loop the minimum is recalculated by evaluating all edges and their weights. For the independent distance metric, this loop is executed $|\mathcal{E}| - 1$ times, as each iteration merges two clusters into one, until a single root node is reached. The calculations for the minima in these loops takes roughly

$$\sum_{j=2}^{|\mathcal{E}|-1} \frac{1}{2} j(j-1)$$

steps, which for the set of 100000 elements amounts to several days on a contemporary personal computer.

A number of intricate algorithms have been proposed to find the closest pair of elements in a multi-dimensional dynamic dataset efficiently, while maintaining low memory consumption, such as [GRSS98]. Others provide optimizations for contexts where the computations of the distance metrics are exceptionally expensive ([Nan05]), which is not the case here. A simpler, more compact solution based on algorithm 1 is as follows. Consider the sequence $\mathcal{C} = p_1, \ldots, p_m$, with $m \leq n$, the heading subsequence of $\mathcal{D}$ of length m. Inputting only the edges corresponding to the distances of the pairs in set $\mathcal{C}$ to algorithm 1, yields subtrees of the final clustering

tree. These subtrees are subsequently replaced by a single cluster corresponding to their roots. This is possible, because of the earlier cluster aggregation choice. As a result, calculations of distances between clusters can be carried out without any knowledge of their internal composition. It depends only on the outline of those clusters. This in contrast to the single-linkage and complete-linkage aggregations.
The roots of the resulting subtrees are subsequently input as a base set to algorithm 1, in the next phase. The internal structure of each subtree is stored nonetheless, but this takes only a fraction of the amount of memory required to store all the edges (for m large enough). By repeating this procedure iteratively, the entire clustering tree is gradually constructed.
Ideally, each iteration of this procedure deals with the approximate number of edges that maximally fit into a user-specified amount of memory M. Assuming the storage of one pair occupies μ bytes, the value of m is now $\frac{M}{\mu}$. The data structure keeping up with the m minimal pairs can then be constructed by iterating over all pairs storing only those of distance $\leq d(p_m)$. This can, for example, be achieved using a sorted datastructure with maximum size m. Once a pair needs to be added when this datastructure has reached full capacity, it is discarded if its weight is larger than the current pair p_i, with $d(p_i)$ maximal in the tree. Otherwise, this maximal pair is discarded and the new pair is added to the sorted datastructure, ideally in $O(log\ m)$ time. Intermediate (during the actual construction of the clustering tree) distances, between newly created clusters and the clusters already in the pairs of $\mathcal{C}$, are inserted analogously. The entire procedure is given in algorithm 4. The larger the value of M, the less often a pair needs to be

Algorithm 4 Build the clustering tree in phases

Require: Set of initial clusters $\mathcal{CL}$, distance metric d_i available memory M
Ensure: V_F and E_F contain the nodes, respectively the edges of the clustering tree F

```
1:  V_F ← CL;
2:  E_F ← ∅;
3:  while 0 < |CL| do
4:     C ← M/μ minimal pairs of clusters ∈ CL sorted by increasing distance;
5:     d_max ← Max(d_i^c(c^j, c^k)) for c^j, c^k ∈ CL;
6:     while 0 < |C| do
7:        Extract minimal pair (a, b) from sequence C;
8:        c ← {a ∪ b};
9:        V_F ← V_F ∪ c;
10:       E_F ← E_F ∪ {c, a} ∪ {c, b};
11:       CL ← CL \ {a, b};
12:       for all (x, x') ∈ C ∧ ((x = a) ∨ (x = b)) do
13:          C ← C \ (x, x');
14:          Recompute d_i^c(x', c) and add pair (x', c) to C if d_i^c(x', c) < d_max;
15:       end for
16:       CL ← CL ∪ c;
17:    end while
18: end while
```

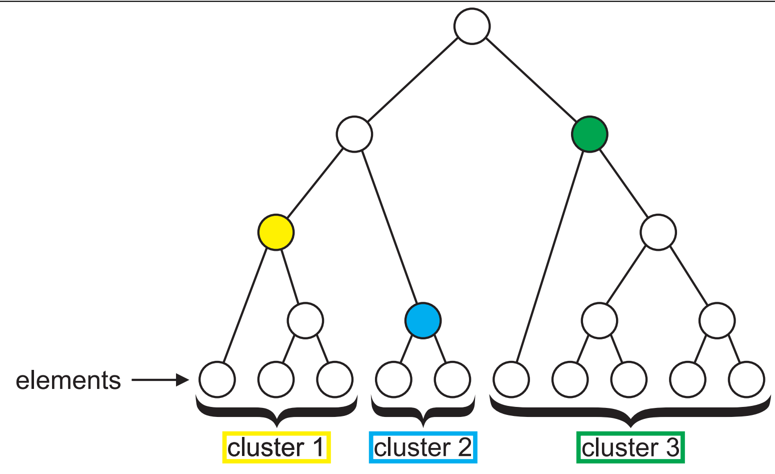
recalculated. The two extremes sketched at the beginning of this section are reached if M can be taken large enough to accommodate all edges in one pass, or if M is taken so small that only a single edge can be considered at a time, respectively. Hence, the trade-off between the speed of the algorithm and the amount of memory at its disposal remains, but through this procedure a descent compromise is reached for fairly large datasets [5].

[5] For the data sets analyzed in this thesis the building of the clustering trees took at most 90 minutes to compute on a 1.8GHz personal computer using 1500 megabytes of internal memory.

3.5.3 Sectioning

This section details on how a *section* in a clustering tree is determined, which will ultimately contain the set of clusters that are drawn to the screen. The discussion includes several diverse methods for traversing a clustering tree using this section.
A *section* is a subset of the nodes in a clustering tree or forest. In the subset of clusters that are drawn to the screen, each member of the base set must be represented exactly once. By definition, a section in the clustering tree is extremely adequate for this purpose, considering that the base set is formed by the leaves of the tree. Figure 3.20 shows a section in a clustering tree and the partitioning it induces on the elements in the leaves of that tree.

Figure 3.20 Section in a clustering tree, formed by the colored nodes, and the partitioning of elements it yields



Stated formally, a section in a tree $T = (V_T, E_T)$, rooted in node $r \in V_T$ is a set of nodes $S \subseteq V_T$ for which the following two properties hold:

$$(\forall n, n' : n \neq n' \wedge n, n' \in S : (n' \notin path\,(n, r))) \tag{3.35}$$

$$\bigcup_{n_i \in S} leaves\,(n_i) = leaves\,(r) \tag{3.36}$$

where for all $n \in V_T$ and root node $r \in V_T$

$$\begin{aligned} path\,(n, r) &= \text{ the nodes on the path starting in node } n \text{ and ending in root node } r \\ leaves\,(n) &= \text{ the leaves of the subtree of } T \text{ rooted in node } n \end{aligned}$$

A section in a forest is the union of the sections of its trees. A section in the clustering forest F is determined by thresholding a user-selected function $\chi : V_F \rightarrow \mathbb{R}$ defined on all nodes of the forest. This occurs by function $\Gamma_\chi : T \times \mathbb{R} \rightarrow S$, with $T = (V_T, E_T)$ a tree and $S \subseteq V_F$:

$$\Gamma_\chi(T, \rho) = \{n \mid n \in V_T \wedge \chi(n) \leq \rho \wedge \rho < \chi(p(n))\} \tag{3.37}$$

Here, $p(n)$ represents the parent node of a node n. For root node r, $p(r)$ is defined as a virtual node with $\chi(p(r)) = \text{inf}$. For this function to be well-defined, function χ must have strictly ascending, or strictly descending values for all nodes on the paths from the leaves of the forest to their roots:

$$(\forall n : n \in V_F \wedge n \neq r : \chi(n) \leq \chi(p(n))) \vee (\forall n : n \in V_F \wedge n \neq r : \chi(p(n)) \leq \chi(n)) \tag{3.38}$$

For function χ, satisfying this property, the section yielded by function $\Gamma_\chi(T, \rho)$, satisfies properties 3.35 and 3.36.
Now, say F is a collection of trees $T_1, \ldots, T_n$. Initially, the threshold value ρ is at its minimum, denoted as $\perp$ and the section contains all the leaves of the forest, or:

$$\Gamma_\chi(F, \perp) = \bigcup_{i=1}^{n} leaves\,(T_i) \tag{3.39}$$

Function $\chi(n)$ for node $n \in V_F$, representing cluster c^i, can be a number of different functions, a few of which are discussed next.

- χ_1 can be the given by the distance, or error level, between a cluster's child clusters:

$$\chi_1(n) = \begin{cases} d^c(c^j, c^k) & \text{if } c^i = \{c^j, c^k\} \\ 0, & \text{if } c^i = \{e^l\} \end{cases} \tag{3.40}$$

  The section resulting from $\Gamma_{\chi_1}(F, \rho)$ follows the order in which the clustering tree was built. Consequently, following the construction of definition 3.34, property 3.38 is inherently satisfied. In most datasets however, the distribution of distances between elements is far from linear. Small changes in the threshold ρ for function $\Gamma_{\chi_1}(F, \rho)$ can lead to the section making large "jumps" in the clustering tree, especially for small values of ρ.

- A more gradual way for traversing the clustering tree in this order is to let χ_2 represent a unique ordering on the clusters by increasing child distance. Clusters with the same child distance are given an arbitrary sequence number. In this case function χ_2 is such that the following hold:

$$(\forall n, n' : n, n' \in V_F \wedge n \neq n' : \chi_2(n) \neq \chi_2(n')) \tag{3.41}$$

$$(\forall n, n' : n, n' \in V_F \wedge \chi_1(n) < \chi_1(n') : \chi_2(n) < \chi_2(n')) \tag{3.42}$$

  If the resulting sequence $\chi_2(n_1) < \chi_2(n_2) < \cdots < \chi_2(n_{|V_F|})$, is chosen such that $\chi_2(n_{i+1}) = \chi_2(n_i) + 1$, traversal of the clustering tree using function Γ_{χ_2} is done linearly in threshold parameter ρ.

- χ_3 can represent the area of c^i:

$$\chi_3(n) = A(c^i) \tag{3.43}$$

  By definition 3.33 a cluster's area is always greater than that of its children, so property 3.38 is satisfied. By traversing the clustering tree according to cluster size, clusters of approximately the same size are drawn together.

- χ_4 can represent the cluster c^i's granularity:

$$\chi_4(n) = G(c^i) \tag{3.44}$$

  where

$$G(c^i) = \begin{cases} G(c^j) + G(c^k), & \text{if } c^i = \{c^j, c^k\} \\ 1, & \text{if } c^i = \{e^l\} \end{cases} \tag{3.45}$$

  Property 3.38 is trivially satisfied. In this way, a section contains clusters with similar numbers of elements.

As a result, through function χ and interactively supplied threshold ρ, the user can explore different levels of detail of the data. Furthermore, through various functions for χ, different subsets of the total set of possible partitionings can be explored. In conjunction with the distance metrics of section 3.5.1, these different sectioning functions offer diverse and flexible ways of traversing several clustering trees, emphasizing a wide range of high-level structures and patterns.

3.5.4 Cluster Coloring

As previously stated, each section in the clustering tree induces a partitioning of the set of elements. The remaining challenge is to represent this partitioning on the screen. Primarily, this can be achieved through a coloring scheme, in which all segments, representing the elements in a cluster are colored identically. In order to easily distinguish between neighboring clusters, it is important that these clusters are assigned a distinct color. Since the human eye can only visually segregate a handful of colors easily, this must be accomplished using as few colors as possible. Only the vertically-adjacent distance metrics, defined on clusters, yield planar subdivisions, which have been proven to be colorable as described above using only four colors. The vertically-independent distance metrics yield non-compact scattered cluster fragments, which generally require a lot more colors. This creates a problem, since the number of distinguishable colors is often much less than the number of colors required to distinguishly color neighboring clusters. One approach is to color a cluster using a color differing from its neighbor's color, where possible and cycle through this small range of colors once they run out. Clusters where this is not possible are then colored with the same color as one of its neighbors. This best-effort approach in practice often fails to distinguish between a large number of clusters. Interactive methods, such as drawing an outline, or displaying the id of a brushed cluster can alleviate this problem, but require extensive and continuous efforts by the user. Furthermore, it takes away from the effectiveness of the clustering technique as a user must first suspect the presence of a pattern or structure. In this situation, the clustering can merely support that intuition. The following section presents a more effective technique for cluster segregation.

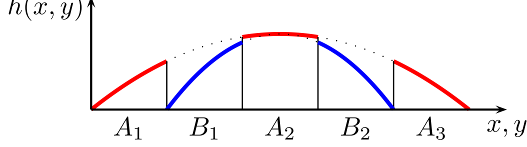
3.5.5 Interleaved Cushioning

Instead of merely relying on color, this section introduces a new method for distinguishing between neighboring clusters by applying a cushion-like texture, similar to those presented in section 3.2. The basic idea is simple. Each cluster c consists of several elements, which in turn are rendered as a set of segments $\{s^i\}$. The bounding box B for these segments corresponds to the projected rectangle induced by the cluster. For consistency's sake, B is represented as two intervals:

$$B = \langle \Pi_X(s^c_T), \Pi_X(e^c_T), \Pi_Y(s^c_L), \Pi_Y(e^c_L) \rangle \tag{3.46}$$

Next, a 2-dimensional parabolic, or plateau cushion profile $h(x, y)$ is constructed, spanning rectangle B. Subsequently, segments s^i are rendered, in which the luminance of a point $(x, y) \in s^i$ are given by $h(x, y)$. Figure 3.21 sketches the idea schematically for two non-compact clusters constraining the disjoint segments $A1, A2, A3$ and $B1, B2$. This technique emphasizes cluster boundaries through dark discontinuities ($h(x, y) \approx 0$). Bright discontinuities ($0 < h(x, y)$) are edges separating intertwined clusters. In conjunction with random cluster coloring and feedback

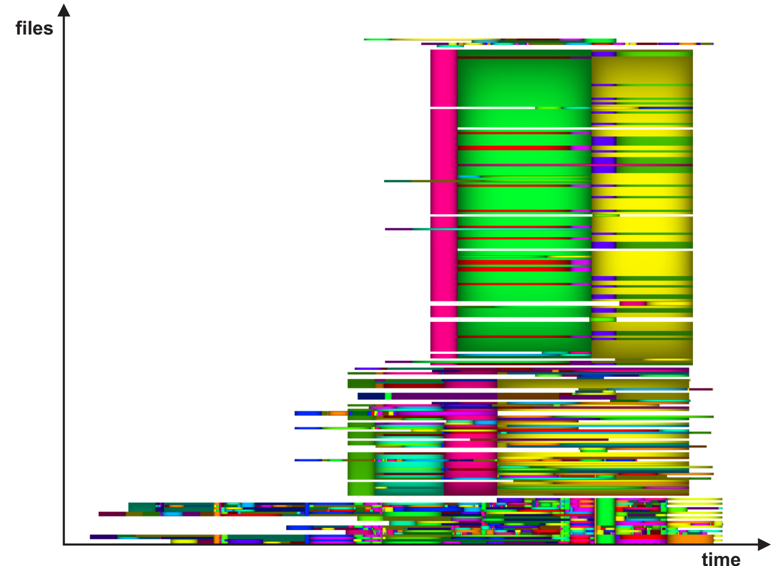
Figure 3.21 Two interleaved cushions: $A_1A_2A_3$ and B_1B_2



on the outline of the cluster, these cluster textures segregate clusters in a visually appealing and effective way. The yield is a set of interleaved cushions as shown in figure 3.22. Despite the non-compactness of the clusters in the middle of this figure, their extent is easy to follow, using the smooth luminance variation of their parabolic cushions. Note that while plateau cushion's

are best to show individual segments (see section 3.2.2), the symmetry and intuitive course of the parabolic cushions is perceived to be far more effective for these non-compact scattered clusters.

Figure 3.22 Interleaved parabolic cushioning applied to clusters in a zoomed in view of a CVS repository of a software project



3.6 User Interaction

This section describes the tool developed that incorporates the techniques described earlier in this chapter. Section 3.6.1 describes some technical details such as the libraries used and the minimum system requirements. It also shortly presents the two main windows of the tool. Section 3.6.2 sketches some of the most common user interface tasks a user might perform during a typical session with the tool. The focus lies on the configuration of the various visual parameters that are found in this thesis. This section is not meant as an exhaustive user manual, but instead attempts to give the reader new to the tool, a general idea of its capabilities.

3.6.1 General Tool Description

The visualization tool that implements the techniques presented in this chapter is written in C++, using the OpenGL API for the graphical portion and FLTK[6] and FLU[7] (extended FLTK widgets) for the user interface. With portability in mind, the choices for the graphical API and user interface libraries are made consciously, as they are operating system independent. In fact, the tool was built successfully on *Windows XP* (32-bit version) and *Windows Vista* (64-bit version), using the MinGW compiler[8] and on both 32-bit and 64-bit versions of *Linux* using the GNU C++ compiler[9]. The minimal system requirements for running the tool differ depending on the number of elements in the input dataset and the specific task that is performed. The main constraint is formed by the system's memory. Two types of tasks are distinguished:

- **Importing**: Initially the dataset is delivered as a log containing data on the software artifact at hand. This data needs to be converted to internal datastructures that commonly

[6]http://www.fltk.org/
[7]http://www.osc.edu/ jbryan/FLU/
[8]http://www.mingw.org/
[9]http://gcc.gnu.org/

represent a set of events specific to the application. These events are subsequently converted to generic datastructures that represent elements. The different clustering trees are then generated from these elements. The generic datastructures are saved to disk. As this operation is bothered with building the clustering trees, it requires large amounts of internal memory, mostly due to the quadratic nature of the algorithms in section 3.5.2.

- **Loading**: Once the raw log data has been imported and saved, it can be loaded and displayed relatively quickly. Data sets containing around 50.000 elements take up roughly 250 megabytes of internal memory.

In general, for smooth operation on moderately large datasets ($\approx$ 50.000 elements), the tool's minimal system requirements are as follows:

- Processor: An Intel®Pentium®IV or AMD Athlon®XP
- RAM: 1024 MB of memory, if importing is required and 512 MB of memory otherwise
- Graphics card: Any card containing at least 32 MB of memory and supporting hardware accelerated OpenGL 1.1

The tool consists of two separate windows, namely one for adjusting the parameters of the visualization and one for the visualization itself. Figure 3.23 shows the first window holding the available controls for the visualization. This window is used for controlling things like the type of cushions, the sub-sampling bias and the clustering section in the visualization. Figure 3.24 shows the visualization of VTK, a large software project. In these figures, the most important components of these windows are labeled.

3.6.2 Tasks

Task 1: The user can import a log file into the visualized data structures by choosing "Import" from the "File" menu and browsing to the desired file. The user is then prompted to provide an indication for the amount of memory available to it. This parameter should be chosen carefully. As noted before, this is a highly memory intensive procedure and the more memory the tool is granted, the faster it will complete its task. Granting more memory then available will however lead to significant slowdown caused by the use of a pagefile. The tool may even crash when this resource also runs out.
When the importation process has completed successfully, the results are displayed in the main visualization window. Furthermore the built datastructures are automatically written to a binary file with extension *.ndf* for later retrieval (See task 2).

Task 2: Alternatively, the user can open a pre-built visualization in a traditional way by choosing "Open" from the "File" menu and browsing to the desired file. The window selector (**A**) now contains the list of visualizations found in the opened file. The first visualization is automatically selected and displayed in the main visualization window.

Task 3: Suppose the user wants to stretch the visualization in order to view a certain section in more detail. Stretching the visualization is achieved by dragging using the right mouse-button. The portion of the visualization currently viewed is shown in the orientation bar (**N**) to the right of the visualization. Next, by dragging using the left mouse button, or by clicking inside the orientation bar, the area of interest is brought into view. Two common zoom-levels have been pre-programmed into the tool. These two preset zoom-levels are:

1. Fully zoomed out. This is the default zoom level
2. Squeeze-fitted. The visualization is zoomed in maximally with all segments still visible inside the main window

Figure 3.23 Overall view of the user controls

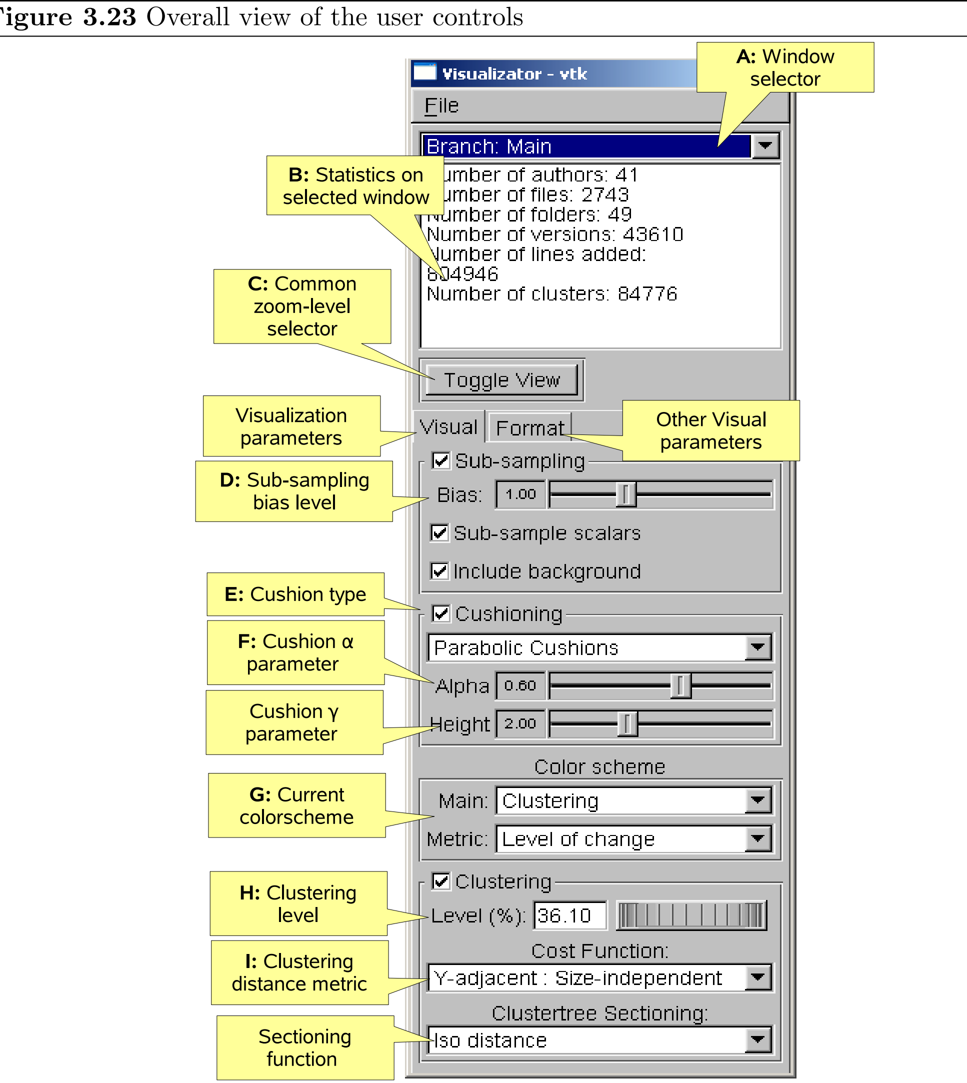
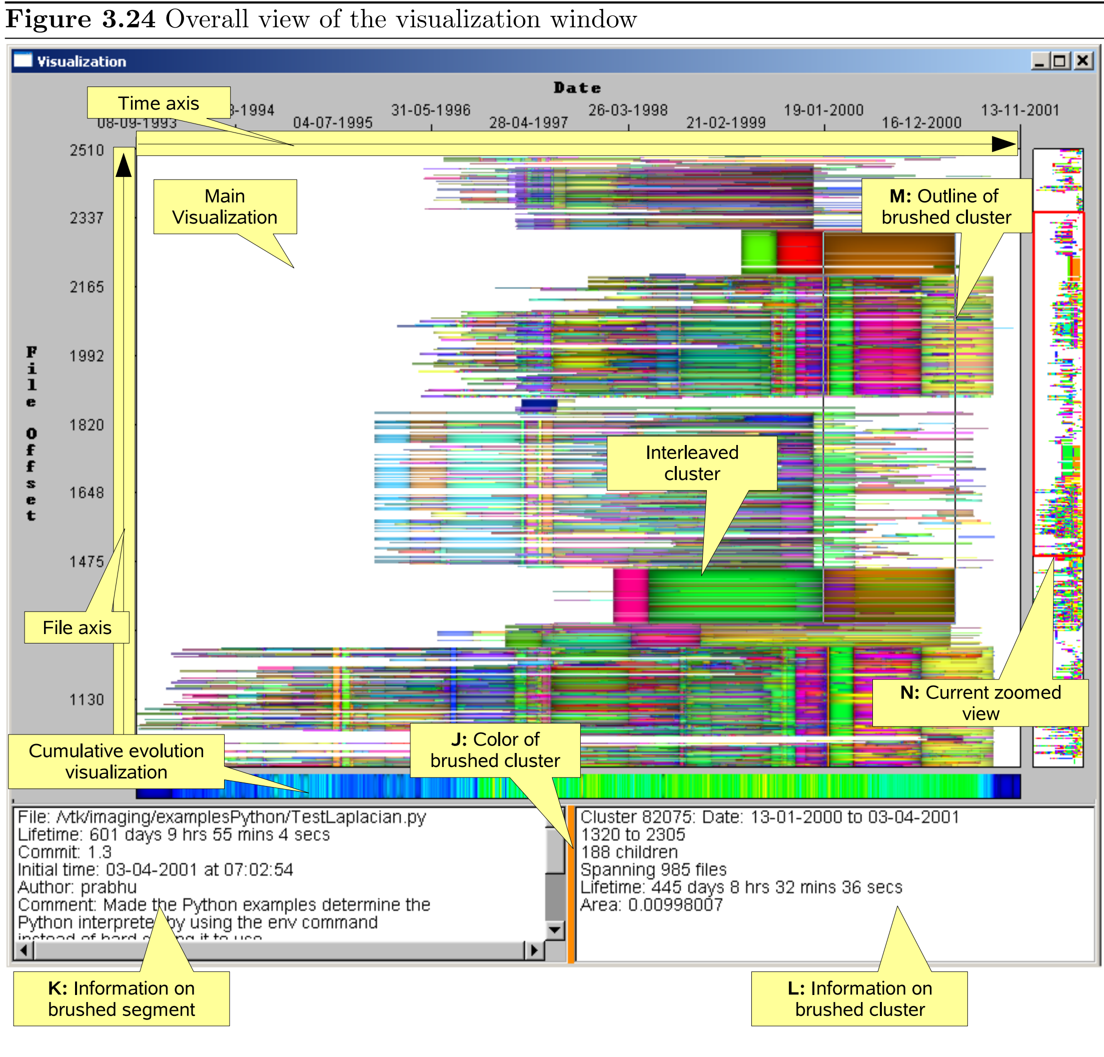


Figure 3.24 Overall view of the visualization window



Button "Toggle View" (**C**) can be used to toggle between the two preset zoom-levels, and the custom zoom-level specified by the user.

Task 4: Suppose the user wants to see the visualization colored by a different attribute. This is achieved by selecting the attribute from menu **G**. Next to coloring by attributes, this menu contains an entry for coloring by clustering. This initially enables the random coloring of clusters.

Task 5: Suppose the user wants to see segments that are smaller than one pixel in arbitrary dimension. This is achieved by lowering the value for "Bias" (**D**) sub-sampling, increasing the priority of smaller segments. Conversely, suppose the user wants to filter out these small segments. By increasing the "Bias" value their priorities decrease. The value for "Bias" corresponds to the α term of the exponential sub-sampling function in section 3.3. A "Bias" level of 1 corresponds to the linear sub-sampling function.

Task 6: Suppose the user wants to cluster the visualization to a certain level. This is achieved by choosing "Clustering" as the current colorscheme (**G**) or enabling it through the "Clustering" check button. This automatically switches the cushion type to parabolic. Now, the level of clustering can be adjusted using input box **H**. The distance metric is chosen through menu **I**. The tool stores the different clustering levels selected for each distance metric. When switching between these metrics the tool updates the clustering level to the corresponding level.

Task 7: Suppose that after having enabled clustering the user wants to see the clusters using the plateau cushions. By choosing "Plateau Cushions" in menu **E** the desired cushions are chosen and subsequently drawn. A third option is to not draw any cushions at all by disabling it. This has the same visual effect as selecting value 0 for the cushion α (**D**). Effectively however, the former option gives faster rendering.

Task 8: Suppose the user wants to view more information on a specific segment or cluster. By brushing the segment or cluster using the mouse, information on this segment or cluster appears in the bottom half of the visualization screen (**J**, **K**). An outline (**M**) is also drawn on the currently brushed cluster so the boundaries of that cluster can be verified. Furthermore the color of the brushed cluster (**J**) is shown to the left of the information on that cluster so that the user can in some sense verify that the brushed cluster is indeed the intended cluster. The cluster information, outline and color verification can be seen in figure 3.24.

Task 9: Suppose the user wants to temporarily suspend a session to continue the analysis at a later time, without losing the current parameters. By selecting "Save" from the "File" menu, all current parameters can be saved to a specified file, which can later be opened again (See task 2).

Chapter 4

Applications and Results

This chapter presents the application of the visualization techniques described in chapter 3 to the two different types of software artifacts described in chapter 2. It attempts to answer the questions that arose in the latter by performing a visual analysis and making a number of observations on the underlying dataset. This is done for two real-life datasets, namely one for the memory allocator application and one for the Software Configuration Management (SCM) system application. Because extensive efforts exist ([VT], [VTvW04], [VT06a]), which analyze SCM systems using techniques that are similar to a number of those described in this thesis, the focus of this analysis will be limited to new techniques introduced, namely importance-based sub-sampling, 2-dimensional clustering and interleaved cushions.
Section 4.1 starts by detailing the dynamic memory allocator application, specifically how allocations can be mapped to segments and what attributes and metrics are assigned to colors. It subsequently explains some of the visual patterns that are interesting for the analysis of fragmentation. It also details some of the most important design decisions that are specific to this application. Section 4.2 then aims at answering the questions posed in section 2.1, by describing various scenarios for the real-life dataset. Similarly, the next two sections describe the SCM system application. Section 4.3 starts by describing a mapping from file versions to segments. The most important design decisions concerning this specific application are presented. A few important patterns, with focus on those yielded by the clustering technique, are outlined. Finally, section 4.4 presents an analysis of a real-life industrial size software project.

4.1 Application A: Dynamic Memory Allocator

An allocation is a fragment of memory allocated for a certain time-interval. Intuitively, a 2-dimensional overview of the behavior of a memory allocator is given by a mapping of memory space against time. The ordering of the y-axis is implicitly given by the memory address space. Since no two allocations can occupy the same piece of memory at the same time, this mapping makes the dataset an adequate candidate, as it satisfies property 2.2. Segments represent allocations of fragments of memory over a period of time.
The mapping from an allocation BA_i of block BL_i in the pool:

$$BA_i = \langle \mathit{begintime}, \mathit{endtime}, \mathit{alloc_proc}, \mathit{dealloc_proc}, \mathit{size} \rangle$$
$$BL_i = \langle \mathit{lowaddress}, \mathit{upaddress} \rangle$$

to element e^i, is as follows:

$$\langle s^i_T = \mathit{begintime}, e^i_T = \mathit{endtime}, s^i_L = \mathit{lowaddress}, e^i_L = \mathit{upaddress} + 1 \rangle$$

Similarly, the mapping from heap allocation HA:

$$HA = \langle \mathit{begintime}, \mathit{endtime}, \mathit{address}, \mathit{alloc_proc}, \mathit{dealloc_proc}, \mathit{size} \rangle$$

to element e^j is:

$$\langle s_T^j = begintime, e_T^j = endtime, s_L^j = address, e_L^j = address + size \rangle$$

The color of a segment for both types of allocations will be determined by categorical attributes *alloc_proc* and *dealloc_proc* and derived continuous attribute internal fragmentation level, which equals:

$$\frac{size}{(upaddress - lowaddress + 1)} \tag{4.1}$$

for allocations in the pool and normalized *size irregularity*, colored using the rainbow color map. Note that visualization of the internal fragmentation level in the heap is useless as the size allocated is always equal to the size requested. *Size irregularity* describes a measure for the "badness" of an allocation's fit in a bin or the heap, based on the block size of the previous[1] bin. Here, a good fitting allocation has size greater than this block size. This is useful for determining whether the allocator allocates bigger blocks than necessary, only when strictly needed. Each process is associated with a single color throughout all bins and heap. This is done for to simplify analysis of inter-bin/heap process correlations. However, as not all processes are likely to be active in all bins, coloring the processes per bin would likely provide a better color identification scheme for each individual bin.
The vertical range of the visualization is the memory space occupied by the bin or heap visualized, to be more precise, $Y_{min} = \Pi_Y(a_1)$ and $Y_{max} = \Pi_Y(a_2 + 1)$, where a_1 and a_2 are the lower- and upper addresses of the bin or heap, respectively. Time ranges from the first allocation made by the allocator in any bin or heap, to the last allocation or deallocation in any bin or heap, to be more precise, $X_{min} = \Pi(t_1)$ and $X_{max} = \Pi(t_2)$ where $t_1 = Min(\{s_T^i\})$ and $t_2 = Max(\{e_T^i\})$. Two different metrics, namely the normalized accumulated occupancy and the normalized accumulated internal fragmentation, can be presented by the metric bar using the rainbow color map. Here high occupancy and high levels of internal fragmentation are intuitively mapped to the red portion of the scale, while low occupancy and fragmentation levels map to the blue portion.
In section 2.1, a number of questions arose:

- What processes require a lot of memory?
- How much space is wasted by allocating more memory than required?
- How is fragmentation distributed in memory?
- Are there fragmentation patterns which can be ascribed to the allocator?
- Are there other fragmentation patterns which can be ascribed to the monitored application?
- Which are the largest quasi-compact regions allocated?

The analysis that follows in section 4.2 will attempt to answer these questions implicitly for the specific dataset or formulate pragmatic clues for answering them in general. First a number of different visual patterns are outlined that implicate specific program or allocator behavior.

4.1.1 Patterns of Interest

In [WJNB95] the importance of the profiling of real programs instead of random simulations is emphasized. Real programs often exhibit strong regularities, which are not easily imitated by synthetic sequences. Some interesting patterns exhibited by real programs, having implications for fragmentation include:

[1]For the heap, the previous bin is the bin with the largest block size, or specifically bin 12. Bin 0 has no previous bin, so all allocations therein have a good fit.

- Ramps: Many programs accumulate datastructures monotonically and gradually over time, after which they discard them again quickly. This results in a gradual incline in the number of blocks occupied, followed by a steep drop to a lower occupancy. A backward ramp occurs when datastructures are quickly built up and are then gradually discarded.
- Peaks: Many programs use memory in bursty patterns. Similarly to ramps, peaks result from building up large datastructures and subsequently quickly discarding them, but peaks are of shorter duration. The distinction is however not precise.
- Plateau's: Many programs build up data structures gradually and then use those datastructures for long periods of time, after which they are discarded, again gradually.

Because of the monotonic and constant behavior of ramps and plateau's, any fragmentation caused in these periods is due to short-term fragmentation. This is convenient for the analysis of fragmentation of the heap. Peaks, indicate a phased behavior of a program, which is the major cause of fragmentation ([WJNB95]). Many objects are allocated and freed, possibly leaving scattered survivors that fragment large areas. This behavior may cause problems for later phases in both the bins and the heap. Consequently, a good general strategy for an allocator is to place objects that will die at around the same time contiguously in memory. This is often achieved by exploiting the following heuristics:

- Objects that are allocated around the same time are likely to die together as well, so allocating consecutive objects contiguously is a simple way to limit fragmentation.
- Objects of different sizes are likely of different types, related to a different activity, so additionally avoiding the intermingled allocation of objects of different sizes can limit fragmentation further.

Ramps, plateau's and peaks can be easily located using the occupancy metric in the metric bar. Ramps occur at points where a monotonic change from cold colors to warm colors, followed by a sharp drop-off back to cold colors, is visible. Peaks result in a quick change from cold to warm colors, quickly followed by a sharp drop-off back to cold colors. Plateau's are visible through a gradual change from cold to warm, followed by a relatively long period of constant color, followed by a gradual change back to cold colors. Alternatively, these patterns can also be located using the main visualization, although they might be harder to identify because of high levels of external fragmentation.
Through the use of clustering, the tool can also emphasize a number of high-level structural patterns. They can provide hints about the activities of the instrumented programs and the behavior of the allocator. These patterns include:

- **Similar groups**: Groups of allocations having approximately the same shape and size could represent a reoccurring activity. It is important that the allocator handles such a repeating task in a similar way.
- **Strips**: A group of allocations that start and end around the same time induce a vertical strip across the visualization. The allocations within these strips are likely to be related to the same data structure, such as a list. Identifying these strips clarifies the datastructures in use by the system.
- **Compact strips**: Allocating similar lifetime allocations, or strips, adjacently in memory, leads to decreased external fragmentation as they are freed at approximately the same time. The compactness of these strips is hence a decent measure for the allocator's ability to predict an allocation's lifetime. Placing long-lived objects together specifically is a good strategy for limiting external fragmentation.

- **Correlations between groups of blocks and attributes**: By comparing the level of correlation between some of the patterns above and attributes such as allocating and de-allocating process id's, size and internal fragmentation, conclusions can be drawn about the behavior of the allocator and the effectiveness of the strategy. If, for example, a lot of non-compact strips occur, belonging to the same process, it could indicate that the allocator policy should provide stricter rules with respect to process id's.

Clustering helps the user in locating these patterns. For example, as the independent metric locates similarities across blocks independently of their vertical position, it is extremely adequate for emphasizing strips of blocks and analyzing their compactness. By switching between cluster coloring and process id, or fragmentation coloring, dependencies between the allocations in these strips can be verified.

4.2 Memory Allocator Application: Results

This section discusses the results of a short visual analysis of a dynamic memory allocator. No definitive conclusions about the allocator should be drawn based on this analysis. The dataset is simply too small and specific, profiling the run of a single program for a short period of time. Conclusive research for allocators serving a broad purpose, like the one in question, should be much more exhaustive, considering a wide range of programs, each having their own allocation patterns. This is beyond the scope of this section and this thesis in general. Instead, this section, demonstrates some analysis scenarios that utilize the techniques developed in this thesis.
Generally, analysis of allocation traces is performed to detect program patterns, uncover how the allocator deals with these patterns, especially concerning fragmentation. The real-life dataset considered originates from a profiled allocator that implements the best-fit mechanism, a sequential fits policy. The basic strategy of this policy is to minimize the amount of wasted space by ensuring that fragments are as small as possible ([WJNB95]). It achieves this by iteratively searching through all entries in a free list to locate the free continuous fragment or block that fits it best, hence the name "best-fit". This policy generally exhibits good memory usage at the cost of poor worst-case performance. Another drawback of the best-fit strategy is that it often finds a very good, but not perfect fit, quickly creating unusable small fragments. With these considerations, this mechanism may not yield the most interesting results for this type of analysis. Instead, a side-by-side comparison of traces for this policy and for a different policy having better performance in general could prove to be useful. The best-fit algorithm allocates blocks in the bins from low to high addresses, while for the heap, it allocates from high to low addresses. No further details of this dataset, such as the specific task(s) carried out during the profiling, are known. The log contains no de-allocation events of unallocated memory fragments, indicating that profiling started together with the allocator. However, a number of allocations are never de-allocated, which could point to memory leaks. The large number of occurrences of this phenomenon however, leads to a strong presentiment that this is not the case and that in fact, the profiler was interrupted prematurely.
This scenario concerns a monitored period of roughly 4 minutes, during which a total of 54 processes issue requests to the allocator. The pool contains 13 bins, subdivided as described in table 4.1 along with some other statistics. The last column of this table also shows the number of allocations made in this particular scenario. In total, the scenario contains 119932 allocations. Bins 0 through 12 and the heap, using the process id coloring scheme are depicted in figure 4.1. This figure also emphasizes the minimal amount of free memory available for each bin or heap, during this scenario. Bins 11 and 12 have periods of full occupancy, while most of the smaller-sized bins have low levels of maximal occupancy. This could indicate a suboptimal subdivision of the particular pool. Furthermore, notice from figure 4.1 how the heap is oriented upside down compared to the bins, which indeed matches the allocator type. The occupancy metric bars for the bins and the heap are also shown separately in figure 4.2. This figure also exemplifies the metric patterns discussed in section 4.1 for this scenario.

Figure 4.1 Squeeze-fitted view of bins 0 through 12 and the heap. The actual occupancy of the bin or heap can be seen in the bar to the right of visualization. The metric bar shows the cumulative occupancy in time

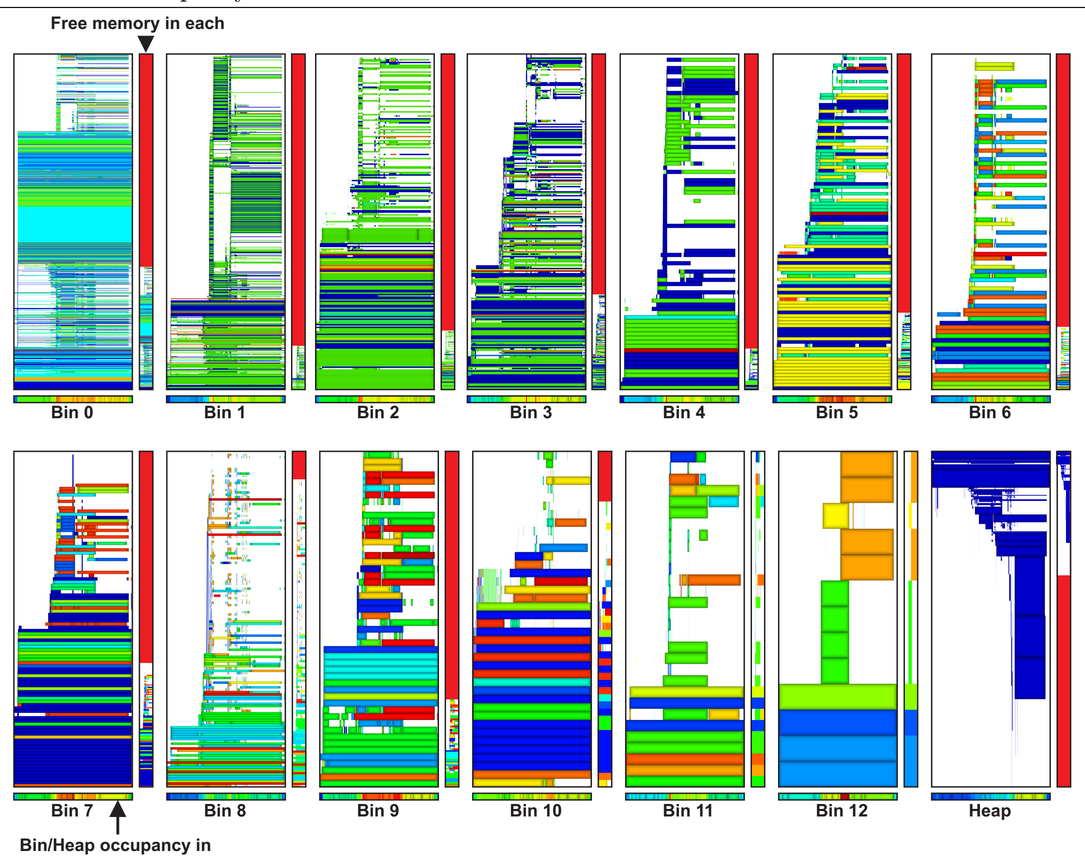


Figure 4.2 Metric bars for bins 0 through 12 and the heap, showing 3 separate phases

Table 4.1: Statistics of the scenario in consideration

Memory	# Blocks	Size of block	Total size	# Active processes	# allocations
Bin 0	3349	16 bytes	53584	52	45728
Bin 1	4158	24 bytes	99792	49	18222
Bin 2	1384	32 bytes	44288	35	8045
Bin 3	910	40 bytes	36400	28	9017
Bin 4	656	48 bytes	31488	22	7864
Bin 5	443	60 bytes	26580	26	6335
Bin 6	415	84 bytes	34860	26	2869
Bin 7	281	116 bytes	32596	22	1588
Bin 8	156	168 bytes	47208	22	7774
Bin 9	191	212 bytes	26208	23	4362
Bin 10	47	284 bytes	13348	23	4697
Bin 11	30	536 bytes	16080	19	1576
Bin 12	13	832 bytes	10816	9	98
Heap	Variable	Variable	711936	15	1757

4.2.1 Phases and Patterns

There is a clear trend directly noticeable from the occupancy metrics in figure 4.2. All bins start with low occupancy, which slowly builds up and reaches its maximum roughly half of the way. This occupancy then gradually drops back to low values at the end of the scenario. The overall trend splits the scenario in three phases, each spanning about a third of the monitored period. It starts off with a period, referred to as *period 1*, of low occupancy which persists until about a third of the monitored period. Despite the low occupancy in this period, there is a lot of activity, especially in bins 0 through 6 and 10. This fact is clearly seen due to the lack of cushions in that period for these bins, which indicates a dense sequence of events. For these bins a lot of short-lived allocations are thus being made, which could lead to high levels of external fragmentation. This is indeed the case for bins 0 through 3, 5 and 6, as confirmed by the sparsely occupied areas in figure 4.3. This figure uses sub-sampling with $\alpha = 0.05$, to emphasize isolated thin segments. Furthermore, notice from figure 4.3 how the high activity areas in bins 4 and 10 (region A and B, respectively), do not directly lead to a higher level of external fragmentation. Instead, many of the short-lived processes in the first period are allocated higher in memory than the long-lived processes. This however, is probably not a conscious decision of the allocator, but most likely just an artifact of the sequence in which the requests arrived. The heap on the other hand is rarely occupied nor active during this period, indicating that little allocations of large size are being made.
After this period, the memory occupancy changes suddenly. This marks the beginning of a period, referred to as *period 2*, of relatively high occupancy, which persists until around two thirds of the scenario. Figure 4.2 also shows that a number of recursive ramps, peaks and plateaus occur scattered across the bins. An overall drop in occupancy (A) splits the period 2 of bins 0, 5, 7 and 9 into two plateau's (B). These patterns however, have little specific implications for the analysis of bin fragmentation, due to the regular structure of a bin. Regretfully, the heap does not show this behavior as explicitly. However, there is a single peak at the end of period 2 and two peaks on a plateau spanning period 3 (See (C) in figure 4.2). Zooming in on these peaks in the main visualization and using brushing, reveals that they are the result of a small number of large allocations (over 10 kilobytes) made for two allocating processes (process 48 for the first peak and 59 for the two later peaks), which leave no scattered survivors (See figure 4.4). Furthermore, note that a number of peaks are filtered out by the bins and consequently, these peaks do not occur in the heap, indeed leading to a more regular occupancy course of the heap.

Figure 4.3 Cropped view of sparsely occupied areas in period 1 of bins 0 through 6 and 10 emphasized through importance-based sub-sampling, colored by allocating process id. In bins 4 and 10, these areas have little effect on the level of external fragmentation

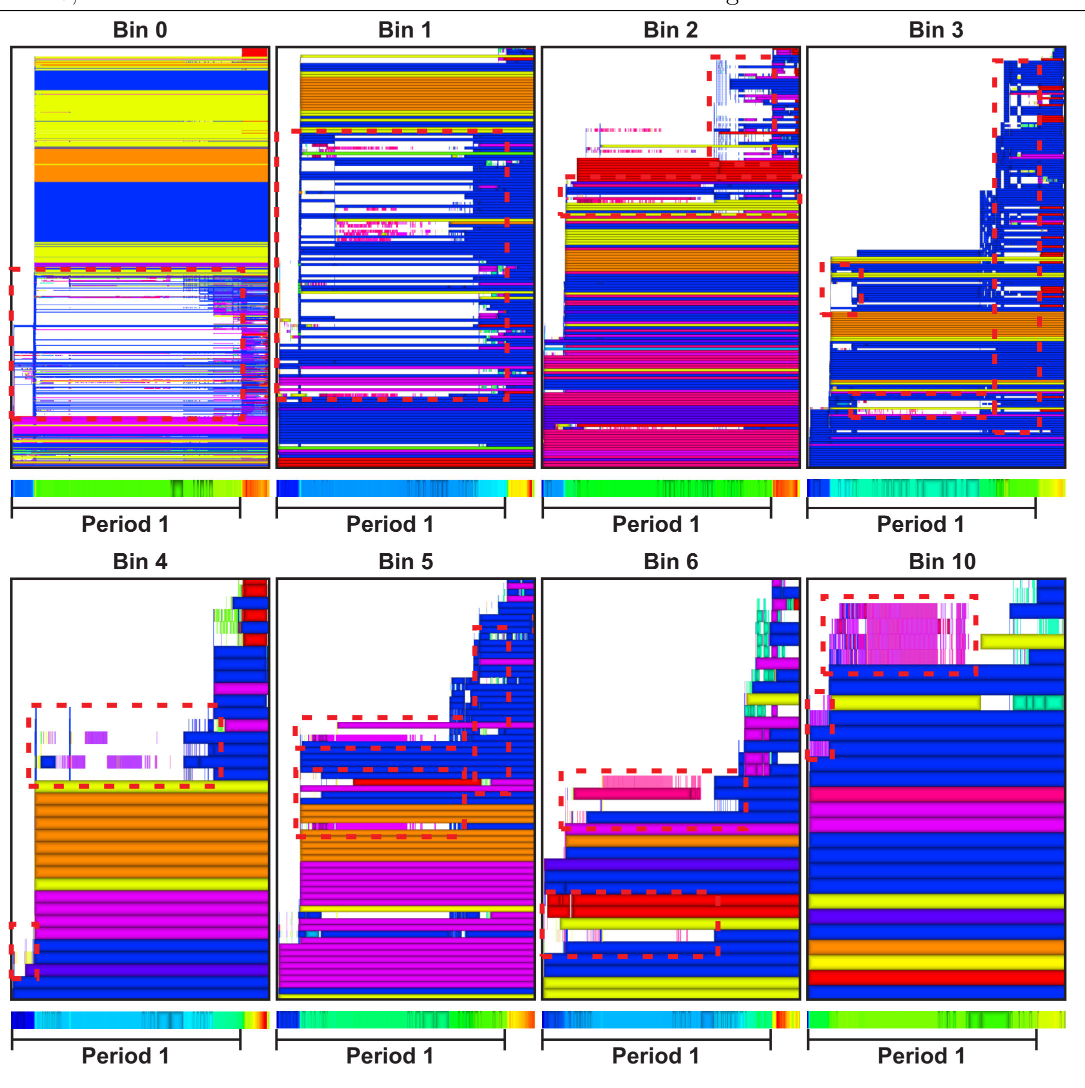
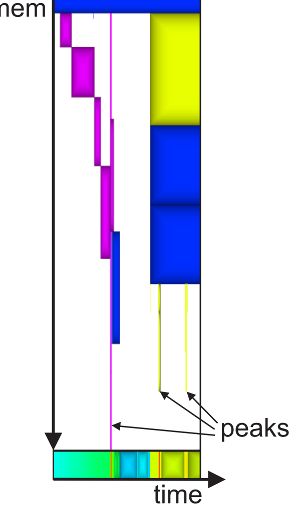


Figure 4.4 Cropped view of three peaks in the heap which leave no scattered survivors

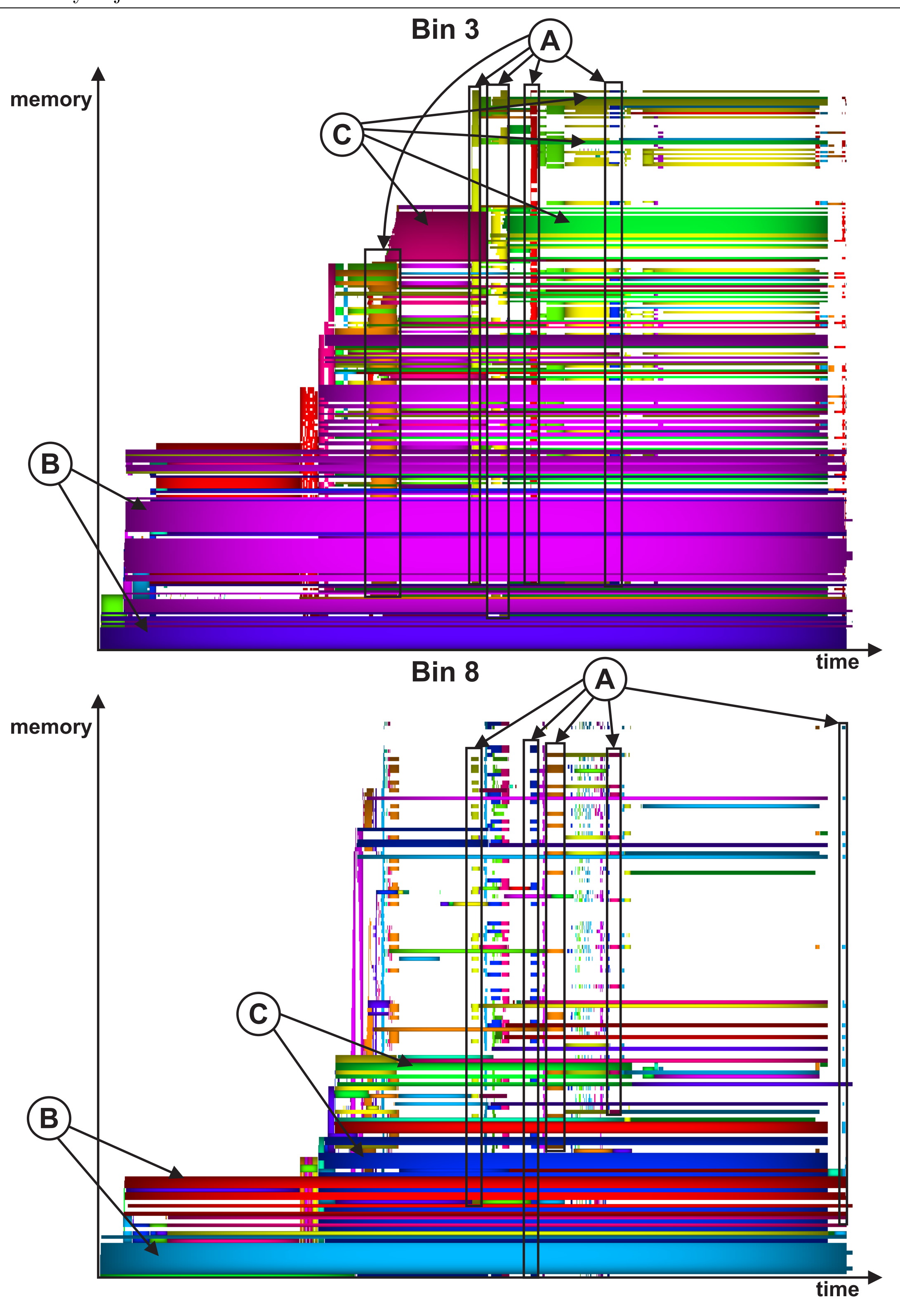


All bins reach their maximum capacity in period 2. The heap however, has a higher general occupancy in the last phase, *period 3*, of the scenario. This is largely due to the three allocated blocks of over 30 kilobytes in this phase. Also notice the larger cushions in the occupancy metric bars of almost all bins and the heap in this last phase, indicating a period of relatively low activity.

4.2.2 Structured Program Behavior

Next to the different program phases, the bins and heap in figure 4.1 exhibit some remarkable similarities in structure. In the following, some of these structures are explored.
To provide insight into the structural decomposition of the program, clustering is applied using the vertically-independent fair distance metric. Because these structures can be of various sizes, sectioning is done by cluster distance. Figure 4.5 of bins 3 and 8 show the most typical structures occurring in all bins and the heap. Clusters A outline some strips of allocations in these bins. A number of these are already apparent in the non-clustered image, although clustering may still reveal their elongated extent (Notice how a lot of the strips in these images start at the bottom of the bin). Others were just not as apparent and are only noticed when clustered. As noted before, these strips, especially those of short duration, are likely to be related to temporary list structures. Traditional array or vector structures are strictly allocated contiguously in memory and are therefore requested as a single contiguous memory fragment by the program. Consequently the structures outlined by clusters A cannot be related to array allocations. Also notice from figure 4.5 how the start of period 2 is marked by a large number of these strips. This seems to be the case for almost all bins, causing the sudden increased occupancy level, noticed in section 4.2.1. Clusters allocations at the bottom of the bins, which span the entire scenario. Their lifetime equals that of the whole process, indicating that these likely contain global or static variables. Clusters C occur as a limited number of similar lifetime blocks across the scenario. They likely hold local function variables. The green fragments outlined by C in bin 3 are actually in the same cluster as the course of the texture implies. This cluster spans a large number of blocks, while only allocating a few of them. This indicates that the allocator does not always provide good locality of reference.

Figure 4.5 Different types of structures in bins 3 and 8 emphasized by clustering, using the vertically-adjacent fair distance metric



4.2.3 Compactness

For analyzing the compactness of the heap, clustering is used. The goal is to determine how well the allocator groups similar lifetime allocations. The vertically-adjacent distance metric are unsuitable for this purpose. Instead, the vertically-independent fair metric is used. Sectioning is performed in order of clustering, hence using the cluster distance. At a reasonable low error level, already a number of clusters start to appear, having high lifetime correlations. A good level of simplification of the heap is shown in figure 4.6. Higher error rates quickly lead to oversimplification and a misrepresented image for the current goal. Figure 4.6 outlines a number of the most noticeable clusters, showing various compactness levels:

- Clusters A and B are moderately to highly compact respectively. As noted in section 4.2.2 they probably hold global or static variables. Cluster A contains three objects of different sizes. Through a glimpse (brushing) at the allocating process id's of these objects, they turn out to belong to different processes. Hence, other than their similar allocation times, the allocator had no clue about the ultimate relation between these three allocations. Cluster B contains two objects of different sizes allocated by different processes. Yet the consecutive requests for these fragments was enough for the allocator to rightfully group them together.

- Cluster C, D and E are less compact, all having a relatively big gap in the middle of two similar lifetime objects. They likely contain variables for local functions, where C contains variables for subfunctions of the function corresponding to the local variables in E. Similarly, such a relation may also exists between clusters D and C.

- Cluster F spans four objects and is interleaved with cluster C. It is furthermore highly non-compact. This cluster likely contains variables for an important local function, due to its long lifetime and the stack of large allocations that are made during its lifetime.

Clearly there is a trend in the evolution of compactness for this scenario and possibly for the best-fit mechanism in general. As external fragmentation aggravates over time, the compactness of groups of similar lifetime allocated objects deteriorates. This in turn can have a negative impact on fragmentation again.

4.2.4 External Fragmentation

As noted in section 2.1.1, due to the static structure of the pool, the impact of external fragmentation on the pool is minimal. It could however affect the speed of some allocators, depending on their mechanisms. The best-fit allocator mechanism however, in combination with the pool structure, only needs to check the first item in its free list for each bin. If it does not fit this entry, it will not fit any other entries in that bin. If it does, no other entry in that bin will fit any better. Hence the block is found in constant time. Updating the free list is also done in constant time, hence, no real performance gains are possible through the limition of external fragmentation. The tradeoff is in internal fragmentation, which is discussed in section 4.2.5.
In the following, the external fragmentation of the heap is analyzed. Here, it can form a real bottleneck, due to the accumulation of small unallocatable fragments. Using clustering, with the vertically-independent fair distance metric, and through similar area sectioning, a number area of high activity, short-lived areas are clustered. These areas are of interest, because they are likely to lead to short-term fragmentation by leaving scattered survivors. Figure 4.7 shows a zoomed in view of three peaks A, B and C. These individual peaks show high levels of short-term external fragmentation. Notice how peak A has a number of scattered survivors including part of the green cluster and the two yellow clusters. The survivors marked by D are placed contiguously in memory, which is generally considered to be good allocator practice. Survivors marked by E however, are segregated from the others and each other, creating a number of small gaps. In peak B, the uppermost of those gaps is likely the reason for the segregation of objects marked

Figure 4.6 Unclustered heap (top) and clustered using the vertically-independent fair metric at an error level of 25% (bottom)

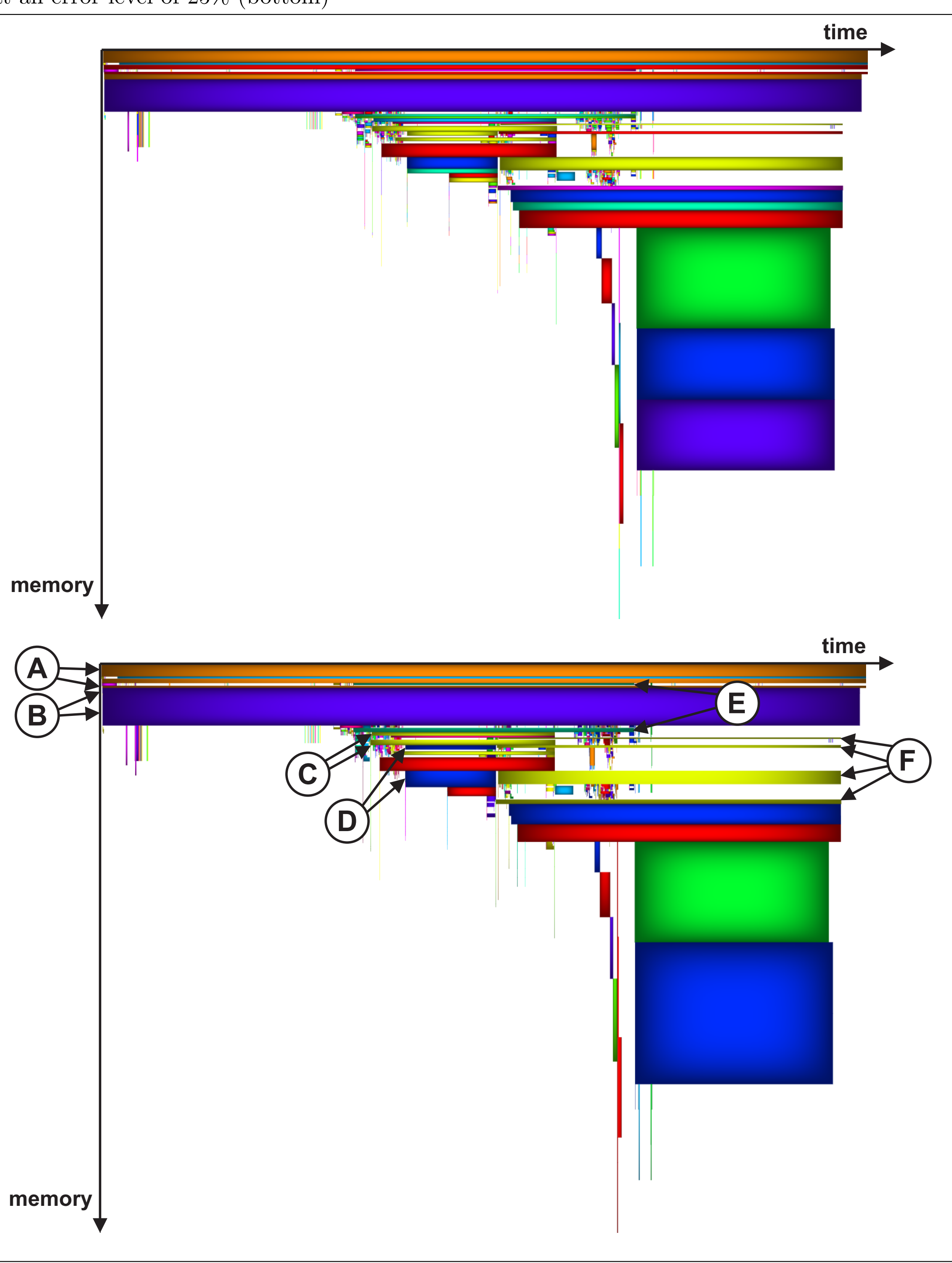
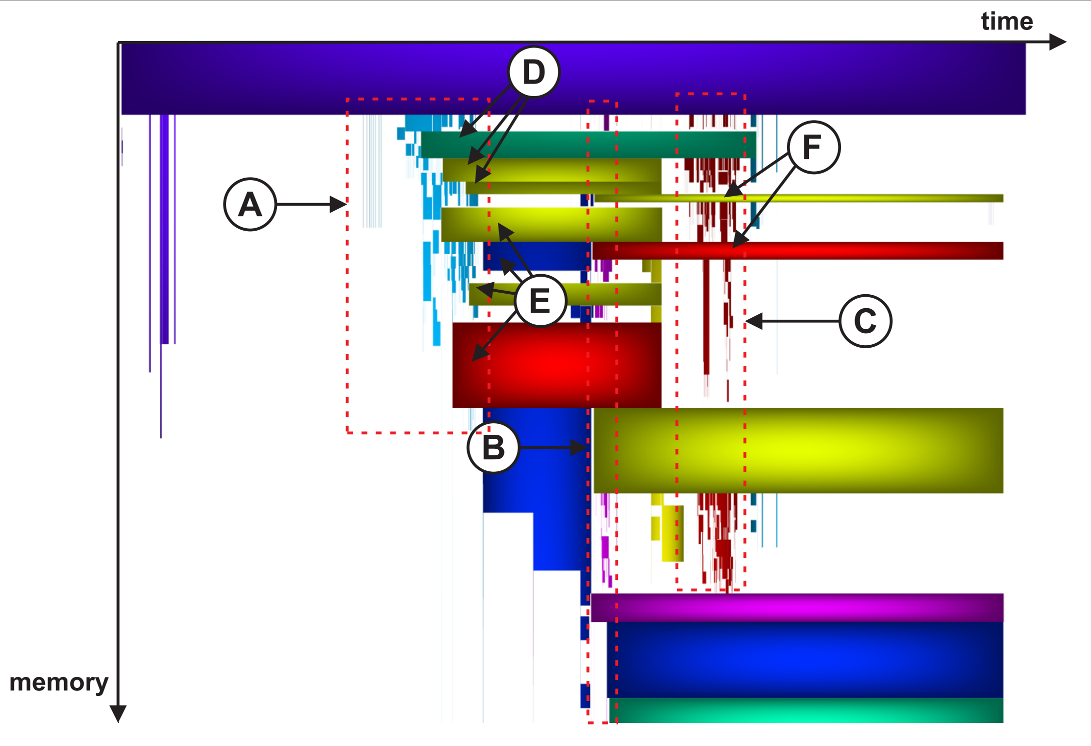


Figure 4.7 Vertically stretched view of the heap showing peaks A, B and C and survivors D, E and F



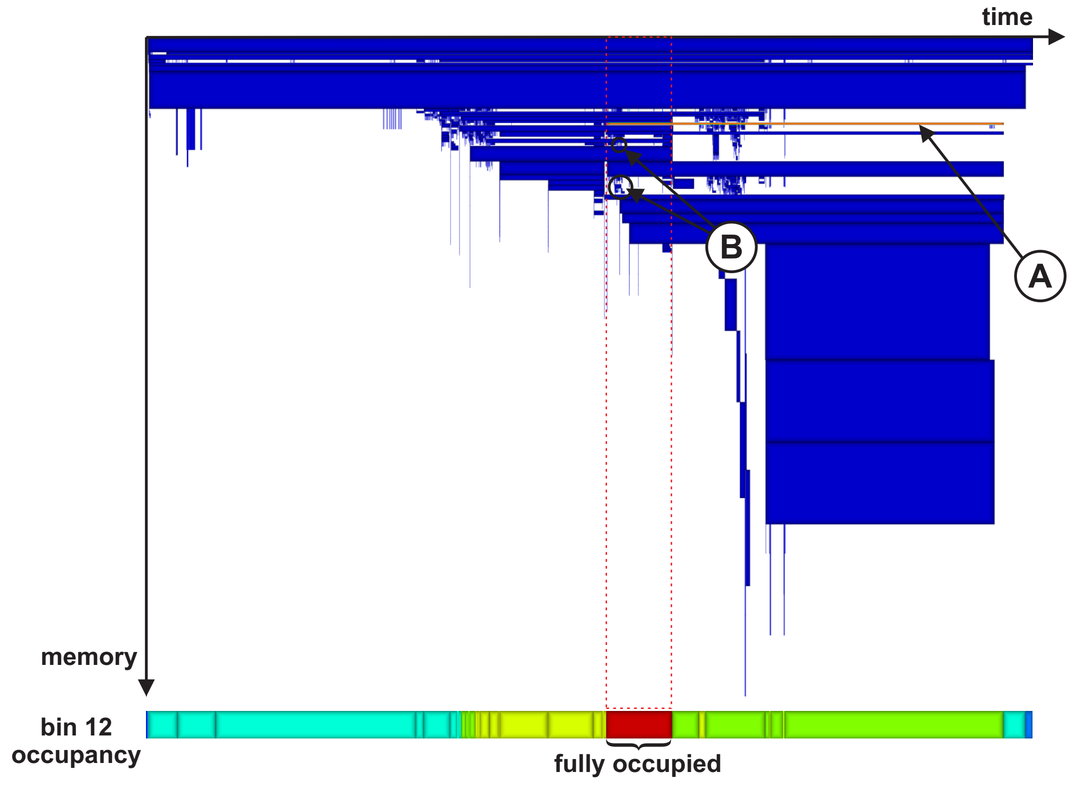
by F. These objects are again scattered survivors, indicating a chain reaction of formation of fragmentation. Peak C exhibits high levels of short-term external fragmentation. It is segregated by several scattered survivors of A and B, but has none of its own. In conclusion note that in this particular short scenario, the best-fit allocator mechanism keeps long-term fragmentation fairly low, as can be seen in the third period of image 4.7. However, as remarked before, this allocation scheme can cause high levels of fragmentation on the long run, due to allocations of near-perfect fit. For the analysis of this type of fragmentation a much longer scenario would be needed.

4.2.5 Internal Fragmentation

For the analysis of internal fragmentation, or waste, the metric bar, displaying the total waste metric can be used. If waste were distributed evenly over all allocations, the waste metric bar would display a similar color course as the occupancy metric bar. Hence, by comparing these two metric bars, periods of increased or decreased waste can easily be spotted. The waste metric bar should exhibit warmer or colder colors in these periods, relating to a higher or lower general waste level. Areas of increased waste in the bins can be due to the occurrence of a phase containing one of the following:

- One or more processes are issuing repeated requests for blocks of an adverse size. Naturally, if such a strip has a requested size different from the average requested size for that bin, the accumulation of a lot of consecutive allocations of this size leads to this deviation. If this occurs frequently for a wide range of programs, it could indicate that the pool structure should be revised to accommodate this adverse size better.
- The allocator is allocating bigger blocks than strictly necessary, either due to an allocator

Figure 4.8 The heap colored by size irregularity (top). One object of irregular size is clearly visible, marked by A. Others are harder to find (B). They are emphasized using sub-sampling $\alpha = 0.05$ The occupancy metric bar of bin 12 is also shown (bottom)



bug or due to a smaller bin reaching full capacity within that period. As noted before, frequent occurrence of the latter for a wide range of programs, could indicate that the pool structure should be revised, such that the smaller bin is assigned a larger number of blocks.

The allocator and also the running program seem to exhibit highly uniform behavior in this respect. The waste metric bar and occupancy metric bar are very similar for all bins. This is probably because not many bins actually reach full occupancy in this scenario. When this is the case, bigger blocks than necessary are likely to be allocated in the subsequent bin, or in the heap, for bin 12, leading to the allocation of smaller objects therein. This is validated through the size irregularity colorscheme. Bin 11 is fully occupied for a fraction of a second, while bin 12 for around 20 seconds. Indeed, only bin 12 and the heap show allocations of smaller size than bin 11 and bin 12's block size, respectively, in the exact periods that these were fully occupied. For the heap, it concerns one allocation with a rather long lifetime as can be seen in figure 4.8. Closer inspection also reveals other short-lived irregular allocations within this period (B). The occupancy metric bar of bin 12 is also shown in this figure to validate that the object of irregular size in the heap is allocated within its fully occupied period. Despite the uniform relation between the occupancy and waste metrics, a few minor deviations occur, of which two are discussed. In figures 4.9 and 4.10, bin 2, 4 and 12 are displayed, in which a number of dissimilarities between their occupancy and waste metrics are outlined. For bin 2, this concerns short peaks, which have a relatively low level of waste (See A and B in the main visualization of figure 4.9). Bin 4 shows the converse. Relatively high levels of waste are reached in the periods indicated by C and D in figure 4.9 (bottom). These are due to a number of subsequent allocations of increased waste. Periods C and D are segregated by a short drop in occupancy. For bin 12 the

Figure 4.9 Bins 2 and 4 colored by internal fragmentation, exhibiting some differences in the occupancy and waste metrics bars shown directly below them

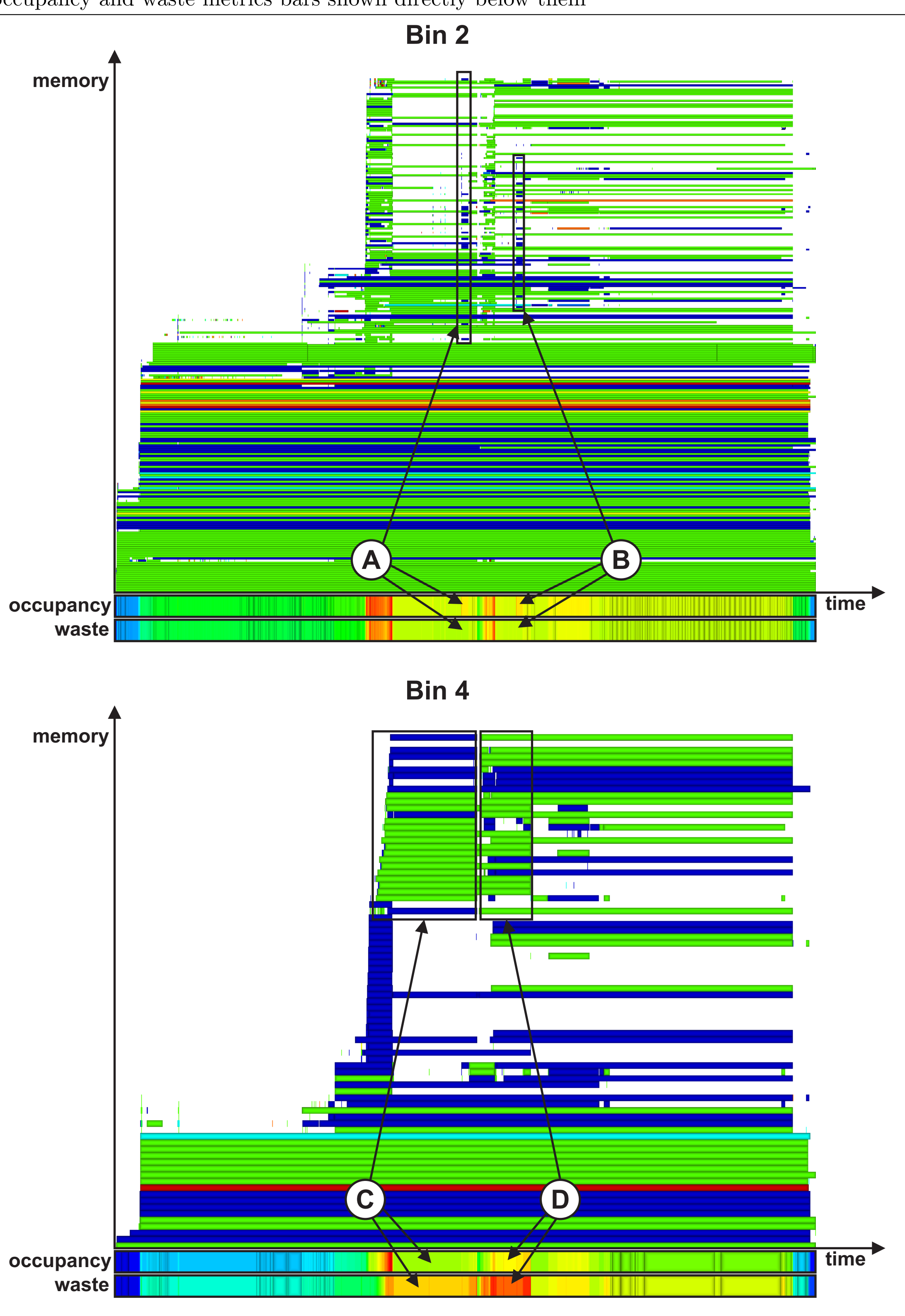
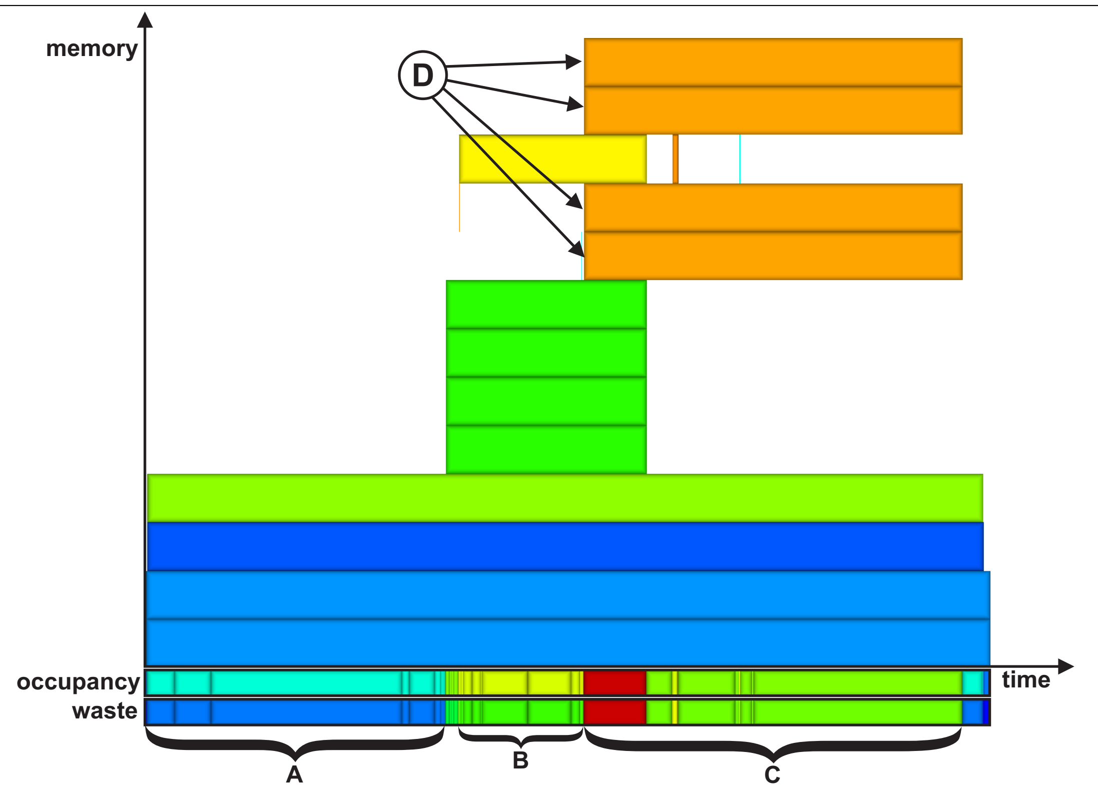


Figure 4.10 Bin 12 colored by internal fragmentation (top), exhibiting large differences in the occupancy and waste metrics bars (bottom)



dissimilarities are more grave in nature. Here, the level of waste is especially large in the second phase of the scenario. This can be noticed from figure 4.10 through the cooler colors throughout A and B, and the lack of this difference in the remainder of the scenario (C). Indeed, looking at the main visualization reveals four large allocations of significantly higher waste in the second phase (D).
Overall, the level of waste in bins 0, 11 and 12 are significantly higher than the level of waste in the other bins, occasionally reaching levels of up to 30% per block. This increased waste level can be explained through the block size ratios of two consecutive bins. The block size ratios between bins 10 and 11, and 11 and 12 are significantly higher. Consequently, bins 11 and 12 accommodate a larger range of object sizes, which directly leading to higher levels of internal fragmentation. Bin 0 suffers from a similar problem, since it accommodates objects from as small as 1 byte to as large as 16 bytes.

4.2.6 Attribute Correlations

Section 4.2.3 already explored some correlations between similar lifetime allocations, emphasized through clustering, and allocating processes. The tool's various colorschemes allows numerous of these attribute correlations to be made. For determining which processes cause high levels of waste, consider figure 4.11. Here, processes 48 and 54 repeatedly allocate strips of objects, a few of which are outlined in the figure. It should come as no surprise that the level of waste is constant within individual strips, as these objects are most likely of the same data structure and consequently have the same size. What is remarkable however, is that all strips belonging to process 48 have the same level of waste, indicating that this process is allocating the same data structure strips numerous times, and hence a correlation between the allocating process and the type of data structure exists. The same holds for process 54. For other processes, like process 28 in figure 4.11, waste levels vary slightly as emphasized by the red arrow. A more exhaustive analysis for this type of correlation would also include different bins and the heap to determine what sizes the processes request in general. This is beyond the scope of this thesis.

4.3 Application B: Software Configuration Management System

For the Software Configuration Management system application, similarly to [VT] file offset is mapped against time. Taking file versions as elements, and by limiting the dataset to the main branch (or trunk) of the repository, thus ignoring sub-branches, this dataset is an adequate one for the methods presented in this thesis. A file can be in only one version at a time in the main branch. Files are ordered by order of appearance in the log, which is created by a depth-first traversal of the repository root. Consequently, files in the same directory get laid out close to each other.
In the following, the definition of a file version of section 2.2.1, is adjusted slightly to accommodate the simultaneous discussion of two such versions:

$$V_{i,j} = \langle author_{i,j}, committime_{i,j}, comment_{i,j}, \#linesadded_{i,j}, \#linesremoved_{i,j} \rangle \tag{4.2}$$

The mapping from a version $V_{i,j}$ to element e^i, is as follows:

$$\langle s^i_T = committime_{i,j-1}, e^i_T = committime_{i,j}, s^i_L = i, e^i_L = i+1 \rangle \tag{4.3}$$

for $j = 2 \dots NV_i$. In this way an element specifies a version of a file as the state of that file between consecutive commit events. The conscious decision is made to relate a version to the commit event that ends it, effectively offsetting all versions by -1. In this way, attributes like author and the number of added or deleted lines actually relate to the time in which the change is carried out. When $j = 0$, the mapping is the same, except for the mapping to s^i_T. This is

Figure 4.11 Bin 8 colored by allocating process id (top) and waste (bottom). Note how the the same processes repeatedly allocate roughly the same sizes

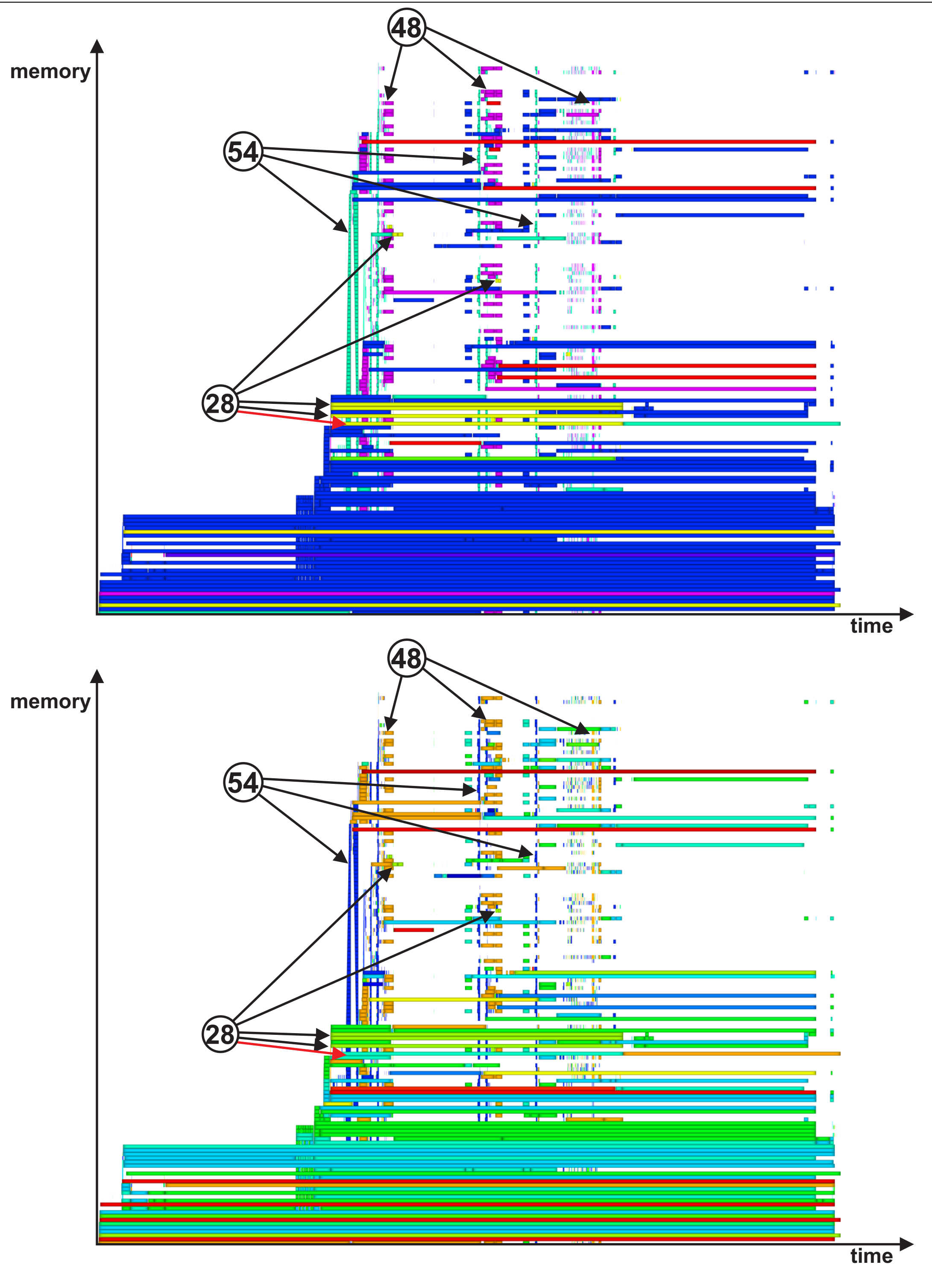
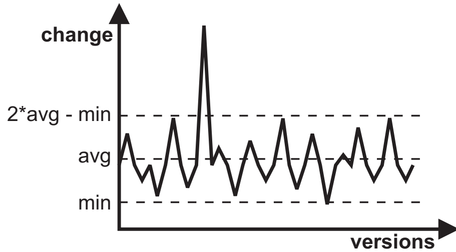


Figure 4.12 Peak change cropping scheme



the first version of a file and no indication of when the efforts for this initial version started is available. A rough estimate is provided by a constant ε, which is a fraction of the total project time. This is not always a very good estimate, as initial versions are often created and committed within seconds, without containing any code. However, as attributes like author name and comment may still be of interest, this version is given a reasonable lifetime. Hence, the mapping for versions $V_{i,0}$ is given by:

$$\langle s_T^i = committime_{i,0} - \varepsilon, e_T^i = committime_{i,j}, s_L^i = i, e_L^i = i + 1\rangle \quad (4.4)$$

The color of a segment will be determined by categorical attribute author name and derived continuous attributes local level of change (normalized per file), given by $\#linesadded{+}\#linesremoved$, and global level of change (normalized for the entire project). Continuous attributes are colored using the rainbow color map. Since the log lacks the number of lines in each version, there is no way of determining the level of change for the initial version. When visualizing the level of change, initial versions will hence always show the color corresponding to the lowest level of change. While it is generally considered bad practice, sometimes binary files are added to a repository. Since the *diff*-tool cannot compare binary files, these files often yield large levels of change. Consequently, the normalized level of change colorscheme loses effectiveness, when the level of change in the source-code files is significantly smaller. When such an extremity is large enough, all segments corresponding to source-code files will exhibit only cold colors in the rainbow color map, while those corresponding with large binary files have highly warm colors. To remedy this problem, peak change values, straying more from the average level of change than the minimal value, are ignore in the normalization (See figure 4.12).
Time ranges from ε time units before the first commit event to the time of the last commit event in the repository. Files, range from 1 to $NF + D$ on the y-axis, where D equals the number of subfolders in the repository. These subfolders appear as empty horizontal lines in the visualization. This can be exploited by clustering using the vertically-adjacent distance metrics, which in this case limits clustering to individual folders.
Next, an analysis is performed which implicitely answers the questions stated in section 2.2:

- How is project-wide activity distributed?
- Which files are heavily modified and by whom?
- Which groups of files are developed together?
- How are these related files distributed over the folder structure?
- At what moments did a mayor release of the project occur?

4.4 Software Configuration Management System Application: Results

This section discusses the analysis of a software project. Some of the techniques presented in this thesis are similar to those discussed in [VT], where these SCM systems are also analyzed. Consequently, this section focuses on the surplus value of the newly introduced techniques, which include importance-based sub-sampling, version-specific clustering and interleaved cushions to visualize these clusters.
The Visualization Toolkit[2] (VTK) project is a popular, large and complex C++ class library containing thousands of files. It is an ongoing project for which coding started early 1994. The log considered in this analysis originates from the CVS repository of this project, mined by CVSgrab ([VT06a]) in November 2001. During this period, 41 different authors contributed to the VTK project that contains 2743 files, the majority of which are source code files. These files are scattered across 49 different folders and have 43.610 versions in total. The following analysis will explore the high-level structure and evolution patterns of the VTK project through global inspection of these versions and files, using a number of different clustering and sub-sampling techniques.

4.4.1 Directory Structure

As mentioned before, clustering using the vertically-adjacent distance metrics, emphasizes structure within the individual folders. When fully clustered, the visualization roughly reveals the directory structure of the project as shown in the top image of figure 4.13. Here, a number of the most important directories are marked. The clusters marked by A, are actually a single directory, namely the “graphics” directory, containing numerous source code files. The clusters marked by B contain “Python” and “CXX examples” directories within the graphics directory. Similarly, clusters marked by C constitute part of the “imaging” directory and clusters marked by D contain “Python” and “Tcl examples”. Clusters E contain the “common” subdirectory.

4.4.2 Evolution Patterns

For locating evolution patterns in seemingly unrelated subdirectories, the vertically-independent distance metric, in conjunction with sectioning by cluster distance can be used. Figure 4.14 shows how this can emphasize, through the interleaved cushioning technique, a high level of structure among versions of files that span the entire project (A). They appear to correspond periods between consecutive releases, due to periods of low activity (stable periods) after a release and periods of high activity shortly before a next release. Comparing these with the actual release moments confirms that this is indeed the case. Also notice how this structure becomes clearer as the project evolves. This indicates that more files are modified in this release phased manner as the project evolves. Furthermore, notice cluster B, which relates two separated subdirectories of the “imaging” and the “graphics” directories. This relation is clearly emphasized by the interleaved cushioning technique. Closer inspection reveals that they are the folders containing the Python examples for the imaging and the graphics directories and that all the files in this cluster undergo the same, very small change. Also notice how the files in these “example” directories do not exhibit the same phased behavior emphasized by clusters A. This indicates that these examples remain unchanged across several releases, which is indeed expected in periods of no major architectural change.

[2] http://www.vtk.org/

Figure 4.13 VTK project fully clustered using the vertically-adjacent distance metric for showing the directory structure

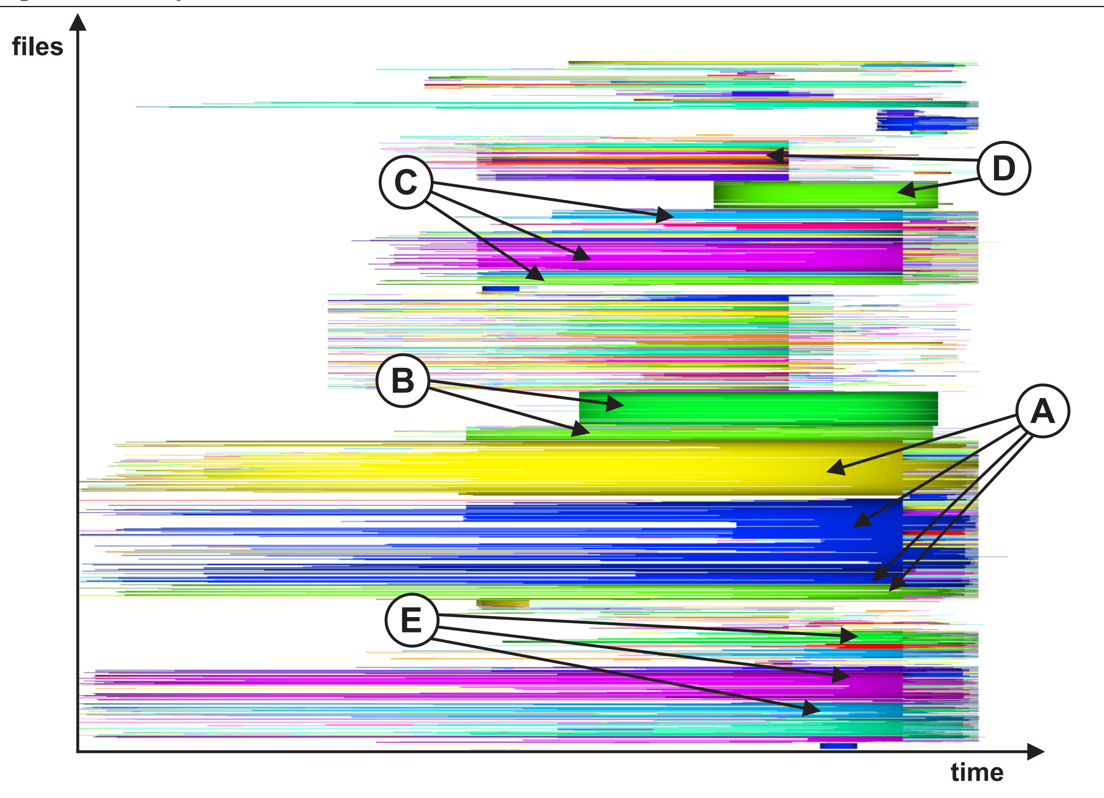


Figure 4.14 VTK project clustered using the vertically-independent distance, revealing similar periodic patterns across the project

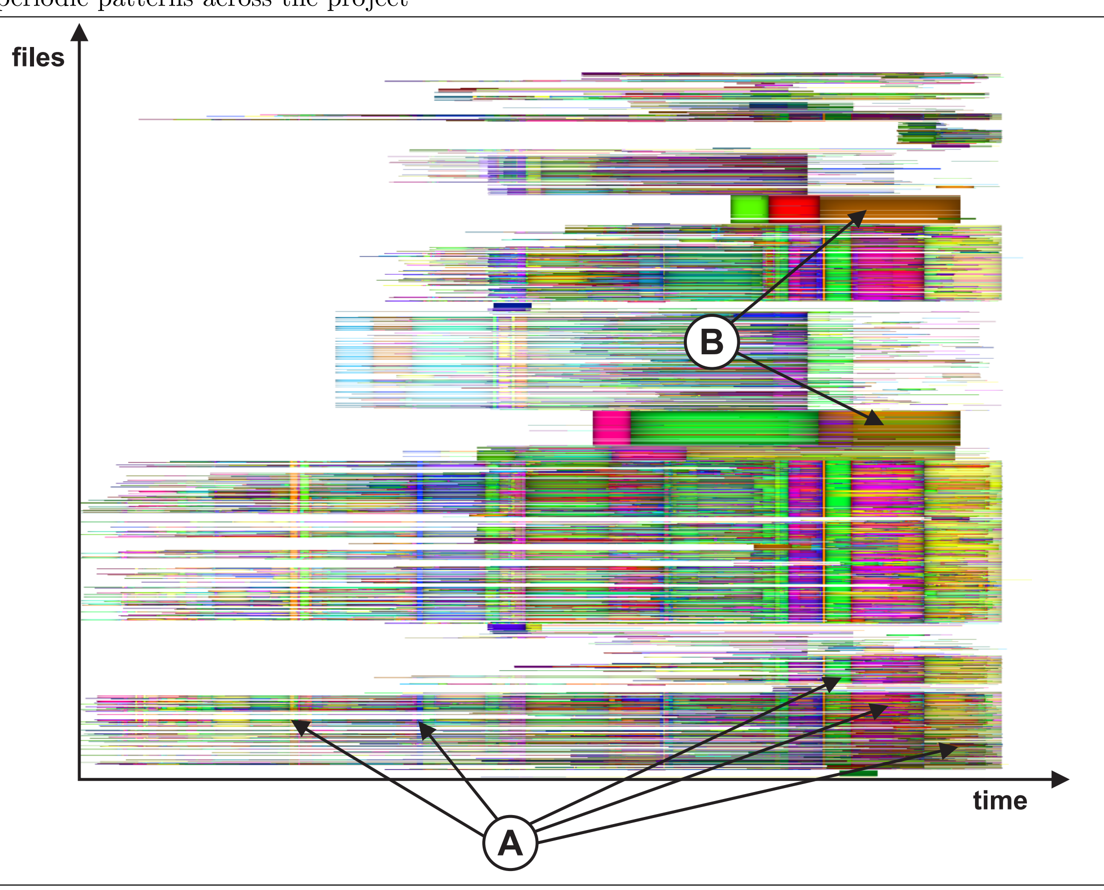
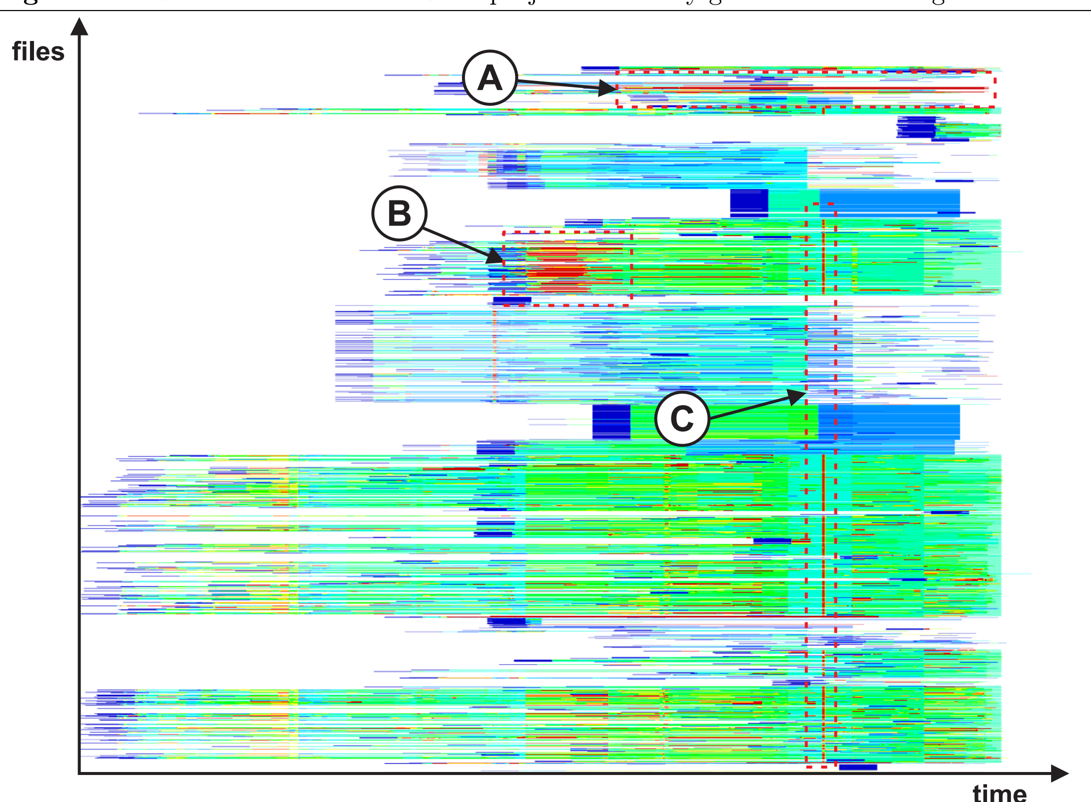


Figure 4.15 Versions of files in the VTK project colored by global level of change



4.4.3 Change Patterns

Typically, during the lifetime of a software project, patterns of major changes occur. The detection and subsequent analysis of these patterns can provide insight into the circumstances that led to these changes and the relevance of these changes to the system architecture. Figure 4.15 shows the familiar visualization of the VTK project, colored by global level of change. A number of interesting patterns are visible. Pattern A shows a small number of files, exhibiting a highly red color, indicating a large level of change. Through zooming in and brushing, these appear to be binary files. As discussed earlier, the *diff*-tool cannot compare binary files. Consequently, they appear completely changed in each commit event, explaining the highly red color in A. By identifying this type of pattern, these binary files can subsequently be removed from the repository. A more interesting pattern is pattern B. Here, major changes are made in a large number of files in the "imaging" subdirectory, during a period of roughly a year. This type of pattern typically indicates an architectural change in a local subsystem. Indeed, for the VTK project, a new API was released for the imaging subsystem in that period. Furthermore, all the major changes in these files were conducted by just two authors, identified by "martink" and "lawcc". Identifying these critical periods can help architects, unfamiliar to the code of an undocumented project, in understanding the most important design decisions. Project managers can quickly determine who were involved in major architectural changes. Finally, pattern C spans across all source code files, denoting a high level of change made in a short period of time. This type of pattern is typically related to project-wide cosmetic activities. Closer inspection of the VTK project reveals that pattern C concerns a change in the copyright notice that is present in every source code file.

Chapter 5

Conclusions and Further Research

The goal of this thesis is to develop methods and techniques that give insight into the evolution of time-dependent software artifacts. Typically the input data-sets represent artifacts from large complex software systems. Several techniques are presented and implemented in tools. These techniques are subsequently validated through two examples from different application domains.

As the basis of the visualization, a dense set of 2-dimensional rectangles are laid out orthogonally, with time on the x-axis. This traditional layout is easily visually invertible and efficiently computable. It provides an intuitive in-depth insight into the given dataset. The basic rendering model applies several colorschemes to the rectangles, for exploring different additional dimensions of the data. This preliminary visualization model is extended by a number of different rendering techniques, including cushioning and sub-sampling.

Cushioning places a parabolic or plateau-shaped texture on top of colored rectangles. The major benefit of this technique is that it increases visual segregation of individual, same-color neighboring segments.

Sub-sampling addresses a problem that is inherent to the large size of the dataset. Limited screen resolution in combination with highly non-uniform and large datasets, quickly leads to rectangles of subpixel size. Sub-sampling copes with this problem by collecting the contributions of all rectangles covering a pixel and combining these appropriately. Furthermore, importance-based sub-sampling is introduced, which biases small rectangles to become more or less visible, based on user preference. Sub-sampling can be performed at two different stages of the rendering pipeline, namely before or after the mapping of an attribute to a color. The former turns out to provide the best results for continuous attributes, while the latter lends best to categorical attributes.

The benefit of the sub-sampling technique is great. In general it gives a much clearer and more detailed picture of dense data. A serious drawback of this technique is performance. Sub-sampling effectively moves the rendering calculations from hard- to software causing a considerable penalty. Apart from this no other serious drawbacks come to light.

An agglomerative hierarchical clustering process groups elements, exhibiting some user-specified level of similarity, together. This is done to support and stimulate user speculations on the higher-level structure of the data. The input to the clustering process is a distance metric which serves as a measure for similarity. A few different distance metrics are proposed including metrics that compare two element's lifetime, size and vertical separation. The trees yielded through these distance metrics are traversed using several sectioning functions, for specifying the level of detail of the visualized data, based on different cluster attributes.

Clusters are primarily emphasized using color, in other words, rectangles belonging to the same cluster have the same color. Not all distance metrics however, yield compact clusters. For more than a handful of non-compact clusters, the number of perceptually distinct colors available for this scheme quickly runs out. As a solution interleaved cushioning is introduced. This is a

simple, yet highly effective technique, which applies a cushioned texture that spans the outline of the cluster to all rectangles belonging to that cluster.

The result of the agglomerative clustering technique, rendered through interleaved cushions is a multi-level, interactive partitioning that, with little user effort, is capable of uncovering a series of otherwise hard to distinguish curiosities in the underlying data.

For validating these methods and techniques, the behavior of a dynamic memory allocator and the evolution of a large software project are analyzed. Through these highly different application domains, the techniques prove to effectively provide insight into the high-level structure concealed in a wide range of dynamic and large datasets.

5.1 Further Research

The techniques presented appear to be effective for a wide range of application domains. Exploring more of these domains is an obvious future direction of this research. Furthermore, the techniques can be extended to provide additional support for visualizing multivariate datasets. Another direction is to further explore the potential of importance-based sub-sampling method, especially related to filtering out unclustered segments. This can enhance the interleaved cushioning technique by removing individual subpixel segments that clutter the partitioned image. Momentarily, the agglomerative clustering methods form the bottleneck on the size of the input dataset of the tool. Through the implementation of more clever and efficient clustering schemes, possibly in combination with the phased tree construction method, larger datasets can be processed by the tool.

Bibliography

[BRT95] Bergman L. D., Rogowitz B. E., Treinish L. A.: A rule-based tool for assisting colormap selection. In *Proceedings of IEEE Visualization '95* (1995), IEEE Computer Society Press, pp. 118–125.

[CM01] Cockburn A., McKenzie B. J.: 3d or not 3d? evaluating the effect of the third dimension in a document management system. In *CHI '01: Proceedings of the SIGCHI conference on Human factors in computing systems* (2001), New York: ACM Press, pp. 443–441.

[CM02] Cockburn A., McKenzie B. J.: Evaluating the effectiveness of spatial memory in 2d and 3d physical and virtual environments. In *CHI '02: Proceedings of the SIGCHI conference on Human factors in computing systems* (2002), New York: ACM Press, pp. 203–210.

[CM04] Cockburn A., McKenzie B. J.: Evaluating spatial memory in two and three dimensions. *International Journal of Human-Computer Studies 61* (2004), 359–373.

[CMS99] Card S. K., Mackinlay J. D., Shneiderman B.: *Readings in information visualization: using vision to think*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[Con03] Conradi R.: Software engineering mini glossary. `http://www.idi.ntnu.no/grupper/su/publ/ese/se-defs.html`, August 2003.

[GRSS98] Golin M. J., Raman R., Schwarz C., Smid M. H. M.: Randomized data structures for the dynamic closest-pair problem. In *SIAM Journal on Computing* (1998), vol. 27, pp. 1036–1072.

[KG05] Keller T., Grimm M.: The impact of dimensionality and color coding of information visualizations on knowledge acquisition. In *Knowledge And Information Visualization* (2005), Springer-Verlag, pp. 167–182.

[LNVT05] Lommerse G., Nossin F., Voinea L., Telea A. C.: The visual code navigator: An interactive toolset for source code investigation. In *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization (INFOVIS'05)* (2005), IEEE Computer Society.

[Nan05] Nanni M.: Speeding-up hierarchical agglomerative clustering in presence of expensive metrics. In *Advances in Knowledge Discovery and Data Mining* (2005), Springer Berlin, pp. 378–387.

[Pre01] Pressman R. S.: *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[Ran69] Randell B.: A note on storage fragmentation and program segmentation. *Communications of the ACM 12*, 7 (1969), 365–372.

[RCM93] Robertson G., Card S., Mackinlay J.: Information visualization using 3d interactive animation. *Commun. ACM 36*, 4 (1993), 57–71.

[RLK92] Rogowitz B. E., Ling D., Kellogg W.: Task dependence, veridicality, and pre-attentive vision: Taking advantage of perceptually-rich computer environments. In *Proceedings of the SPIE Symposium* (February 1992), vol. 1666, pp. 504–513.

[RT93] Rogowitz B. E., Treinish L. A.: Data structures and perceptual structures. In *Proceedings of the SPIE Symposium* (February 1993), vol. 1913, pp. 600–612.

[SBP97] Stasko J. T., Brown M. H., Price B. A.: *Software Visualization*. MIT Press, Cambridge, MA, USA, 1997.

[Str00] Stroustrup B.: *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Swe86] Sweller J.: Cognitive technology: Some procedures for facilitating learning and problem solving in mathematics and science. *Journal of Educational Psychology 81* (1986), 457–466.

[TLTC05] Termeer M., Lange C. F. J., Telea A. C., Chaudron M.: Visual exploration of combined architectural and metric information. In *Proceedings of VISSOFT 2005* (2005), IEEE Press, pp. 21–26.

[VT] Voinea L., Telea A. C.: Visual data mining and analysis of software repositories. *To appear in Computers & Graphics, Elsevier, 2006.*

[VT06a] Voinea L., Telea A. C.: Cvsgrab: Mining the history of large software projects. In *Proceedings of IEEE EuroVis 2006* (2006), IEEE Press, pp. 187–194.

[VT06b] Voinea L., Telea A. C.: Multiscale and multivariate visualizations of software evolution. In *SoftVis '06: Proceedings of the 2006 ACM symposium on Software visualization* (New York, NY, USA, 2006), ACM Press, pp. 115–124.

[VTvW04] Voinea L., Telea A. C., van Wijk J. J.: Ezel: a visual tool for performance assessment of peer-to-peer file-sharing network. In *INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'04)* (Washington, DC, USA, 2004), IEEE Computer Society, pp. 41–48.

[vWvdW99] van Wijk J. J., van de Wetering H.: Cushion treemaps: Visualization of hierarchical information. In *INFOVIS '99: Proceedings of the 1999 IEEE Symposium on Information Visualization* (Washington, DC, USA, 1999), IEEE Computer Society, p. 73.

[WJNB95] Wilson P. R., Johnstone M. S., Neely M., Boles D.: Dynamic storage allocation: A survey and critical review. In *IWMM '95: Proceedings of the International Workshop on Memory Management* (London, UK, 1995), Springer-Verlag, pp. 1–116.