Real-time Garbage Collection of a Functional Persistent Heap

Licentiate's Thesis

Kenneth Oksanen

Teknillinen korkeakoulu	Helsinki University of Technology
Tietotekniikan osasto	Faculty of Information Technology
Tietojenkäsittelytekniikan laitos	Department of Computer Science

Otaniemi 1999

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF LICENTIATE'S THESIS

Author:	Kenneth Oksanen		
Name of the thesis:	Real-time Garbage Collection of a Functional Persistent Heap		
Date:	June 11, 1999	**Number of pages:**	106
Department:	Faculty of Information Technology	**Professorship:**	Tik-106
Supervisor:	Professor Eljas Soisalon-Soininen		
Instructor:			

Traditional database management systems perform updates-in-place and use logs and periodic checkpointing to efficiently achieve atomicity and durability. In this Thesis we shall present a different method, *Shades*, for achieving atomicity and durability using a copy-on-write policy instead of updates-in-place. We shall also present index structures and the implementation of *Shines*, a persistent functional programming language, built on top of Shades.

Shades includes real-time generational garbage collection. Real-timeness is achieved by collecting only a small part, a generation, of the database at a time. Contrary to previously presented persistent garbage collection algorithms, Shades has no need to maintain metadata (remembered sets) of intra-generation pointers on disk since the metadata can be reconstructed during recovery. This considerably reduces the amount of disk writing.

In conjunction with aggressive commit grouping, efficient index structures, a design specialized to a main memory environment, and a carefully crafted implementation of Shines, we have achieved surprisingly high performance, handsomely beating commercial database management systems.

Keywords: Main memory databases, persistent programming languages, garbage collection, byte code interpreters, compilers.

TEKNILLINEN KORKEAKOULU LISENSIAATINTYÖN TIIVISTELMÄ

Tekijä:	Kenneth Oksanen		
Työn nimi:	Funktionaalisen keskusmuistitietokannan tosiaikainen roskankeruu		
Päivämäärä:	11. kesäkuuta, 1999	**Sivuja:**	106
Osasto:	Tietotekniikan osasto	**Professuuri:**	Tik-106
Työn valvoja:	Professori Eljas Soisalon-Soininen		

Perinteisesti tietokannat suorittavat päivitykset suoraan muutettaviin tietoalkioihin ja käyttävät lokeja ja ajoittaista vedostusta (checkpointing) atomisuuden ja vikasietoisuuden takaamiseksi. Tässä työssä esitetään vaihtoehtoinen menetelmä, *Shades*, jonka perusperiaatteena on tehdä muutokset aina uusiin tietoalkioihin. Työssä esitetään myös Shadesin päälle toteutetut hakemistorakenteet ja persistentti ohjelmointikieli *Shines*.

Perinteisistä lokijärjestelmistä poiketen, Shades suorittaa myös roskankeruuta. Roskankeruu on sukupolvittainen ja kompaktoiva. Roskankeruu on saatu tosiaikaiseksi niin, että se kerää kerrallaan roskat vain tietokannan pienestä osasta, yhdestä sukupolvesta. Toisin kuin aikaisemmat levyvarmennetut roskankeruumenettelyt, Shadesin ei tarvitse pitää levyvarmennettuna metadataa, ns. muistojoukkoja (remembered sets), sukupolvien välisistä osoittimista, vaan se osataan rakentaa uudestaan tietokannan toipumisen yhteydessä muutenkin levylle kirjoitetusta datasta. Tämä oleellisesti vähentää levyliikennettä, mikä taas tehostaa tietokannan toimintaa. Kun lisäksi harjoitetaan ryhmäsitoutumista, käytetään tehokkaita hakemistorakenteita, ja Shinesin toteutus on suunniteltu huolellisesti, on koko järjestelmästä tullut erittäin tehokas, jopa kaupallisia relaatiotietokantoja huomattavasti tehokkaampi.

Avainsanat: Keskusmuistitietokannat, persistentit ohjelmointikielet, roskankeruu, tavukooditulkit, kääntäjät.

Preface

In the end of 1995 a three-year research project, Shadows, ended at Helsinki University of Technology (HUT). This project studied and improved a database recovery method known as shadow paging and implemented a database server based on it [94]. In the subsequent year, 1996, we started the HIBASE-project, whose vague plans included implementation of index structures on top of the shadow paging algorithm now rewritten in a new startup company. In the meantime, while waiting for the rewrite, frustratingly little concrete work could be done in the HIBASE-project. Moreover, in February I caught an annoying flu. Under the influence of this virus and some relaxing refreshments that I used to heal myself, I began to rethink the various aspects of Shadows, particularly in a main memory environment. Several mind-boggling revelations, numerous feverish hacking sessions, and two prototypes later we had the first working version of Shades, our garbage collector, and plans for the rest of HIBASE: we were going to build a persistent programming language on top of Shades.

The spring of 1996 was also the time of forming the HIBASE team. At HUT, Kengatharan Sivalingam contributed most of the benchmark programs for HIBASE, Antti-Pekka Liedes wrote the IO and networking code, and Sirkku Paajanen began her extensive work on index structures [67]. They also wrote parts of the first prototype compiler for the byte code interpreter which I had been working on in the winter 1996 to 1997. Lars Wirzenius joined us in the beginning of 1997 to implement strings, maintain some of our computers, and impose some quality on our code.

At Nokia Telecommunications (NTC) Matti Tikkanen and Jukka-Pekka Iivonen had been developing trie-based index structures and studying type systems for some time. Tikkanen also took care of a large portion of the project bureaucracy. In the summer 1997 the project members moved to common premises; such a close collaboration between the university and the industry has rarely been seen.

In the winter of 1997–1998 serious work had began on the database programming language. The language was originally baptized Nolang, mimicking Ericsson's Erlang, but it has since been renamed Shines, which is a recursive acronym for "Shines is not extended SQL". I wrote the core compiler and several optimizations, Sivalingam wrote portions of the peephole optimizer and Liedes wrote function inlining. Two new members joined the team at NTC: Petri Mäenpää came to work on static typing and Vera Izrailit on pattern matching and LR(1) parsing. In the meantime Iivonen had been implementing smart pointer interfaces from C++ and Java to Shades.

The Shines compiler was able to bootstrap itself in May of 1998. Paajanen and Sivalingam had left for new challenges while Marko Lyly and Jarkko Lavinen joined the teams of HUT and NTC, respectively. Lyly has been working on a profiler and debugger and Lavinen on index structures and new optimizations in the compiler. In addition to various optimizations and libraries, explicit threads and continuations were added to Shines in 1998. An exceptional student group consisting of Antti-Pekka Liedes, Ville Laurikari, Sami-Pekka Haavisto, and Joni Niemi implemented a tool for building graphical user interfaces with Shines.

As can be seen, a considerable implementation effort has been invested in HIBASE, and indeed most of its subsystems are highly optimized and have had empty buglists for years despite of continuous exercise. Our serious intention and keen desire is to see HIBASE used to build reliable and efficient commercial products within a few years. I thank my teammates without whom we would hardly stand a chance of implementing a system so wide and of such impeccable quality.

Although Shades was conceived in the spring of 1996, not until May 1997 would we implement recovery. I am grateful to our funder, Nokia Telecommunications, that it never lost its belief in HIBASE despite that, or because of fear of the major undertakings ahead. Lauri Melamies, Keijo Olkkola, Kai Kurru, Maaret Karttunen, Hannu Hellstén, Raimo Kantola, Heikki Saikkonen, Asko Suorsa, and Matti Tikkanen among others supported our work. The project was also funded by Helsinki Graduate School of Computer Science (HeCSE) and the Technology Development Centre (TEKES).

Although we have all been incredibly fascinated with the technology of HIBASE, we would hardly have been sufficiently motivated to keep up with the consuming work without the mutually satisfactory intellectual property rights arrangements negotiated by Tero Kivinen and Raimo Kantola.

Samuli Honkanen and Riitta Karhumaa from Compatent Inc. took care of the patenting of relevant parts of HIBASE and were simultaneously among the first readers of this Thesis. I have also ruthlessly used my fellow team members, most notably Matti Tikkanen and Vera Izrailit, as my proof readers. Also Jari Malinen, Nora London, and especially my supervisor, professor Eljas Soisalon-Soininen, have given me valuable comments. I am most grateful to professor Soisalon-Soininen that he has never lost his belief, or at least not his patience, in spite of my wild revelations and long nights of hacking instead of writing articles or this Thesis.

Contents

List of Figures

Chapter 1

Introduction

The traditional image of a database is that of a large amount of mainly disk-resident data and a server which processes efficiently and reliably various SQL transactions, such as money transfer orders and account balance queries. Current database technology has indeed been very successful in this role, and there is no reason to believe it will be imminently replaced by a different technology.

However, there is demand to utilize database-like features in domains where traditional off-the-shelf database products are considered suboptimal or even insufficient.

- Main memory databases are feasible in increasingly many applications, particularly when high performance is required. Companies can easily spend a few kilobytes of RAM for each employee, product line, or customer. Databases now kept on disk can be kept in main memory in a decade.

  A main memory setting emphasizes some aspects, such as fast query execution, efficient disk writing and aggressive commit grouping, while some aspects, such as fine granularity locking and clustering, become less important [25].

  Except for the size of the database, main memory databases may not compromise any of the requirements pertinent in their disk-based counterparts. Specifically, main memory databases must guarantee durability of committed data.

- Soft or hard real-timeness requirements have to be met in many embedded applications, e.g. in telecommunications or industrial process control. It is hard to provide absolute guarantees on meeting transaction deadlines with traditional database protocols, concurrency control and unpredictable disk IO costs [86, 44].

- The database queries and clients are traditionally written in different programming languages with different type systems, data representations, recovery and replication behavior, and naming and binding schemes. These discrepancies, often called "impedance mismatch", result in considerable amounts of redundant code, typically 30% of the total [9], for translating information from the database representation to the programming language's representation and vice versa.

In an *orthogonally persistent* system, data can be manipulated in a manner independent of whether it is persistent or transient.

- The expressiveness of the relational data model in general and SQL in particular is limiting in many practical cases. While SQL does have means to iterate over relations, computing, e.g., the transitive closure of a relation is impossible in the original SQL. Although numerous extensions to SQL have since been proposed to solve the transitive closure problem [17] and many situations arising in process control [40, 14], SQL still remains a computationally incomplete language. Writing an SQL or LISP interpreter in SQL is impossible.

  In the meantime, more expressiveness, preferably computational completeness, or "Turing-completeness", is needed, e.g. in CAD, artificial intelligence, Internet services, and agent-based computing. Reaching computational completeness also has the benefit of solving the impedance mismatch problem and avoiding the development and performance overhead caused by it.

1.1 Claims

In this Thesis we shall present *Shades*, a new recovery and garbage collection algorithm for main memory databases, and *Shines*, a persistent programming language designed and implemented on Shades. We claim our system to be competitive, if not superior, to previously presented technologies in the domain of real-time main memory databases and database programming languages.

The very basic, and very pervasive, idea in Shades is that updates to database objects are done "copy-on-write". "Updates-in-place", also called mutations and destructive updates, are generally not allowed.

For example, assume we are about to insert a new binary tree node as a child of another binary tree node which is already in the tree. Since we are not allowed to modify the old node, we make a copy of the old node and store in the copied node the pointer to the new child node. But since the copied node's location in memory has changed, we also have to update the pointer to it by performing a copy-on-write update for its parent, and continue similarly to the root of the binary tree.

At first the strict copy-on-write principle may sound overly restrictive and cumbersome, but an entire programming culture, the "functional programming" paradigm, is based on the concept of not allowing mutation. The functional programming paradigm is often advocated to improve code readability, reusability, correctness and provability by ensuring "referential transparency".

Opponents of functional programming criticize it for inefficiency, but in Shades the very contrary is true. The copy-on-write policy has turned out to allow a different, more efficient recovery method with no logs, no checkpointing and no random disk accesses needed to keep the disk image in sync with the mutated main memory image. The copy-on-write principle was an elective decision made early in the algorithmic planning of Shades: it allowed more "design space" for the garbage collector and some other useful mechanisms than if we allowed arbitrary mutation anywhere in the database.

There are only a few carefully selected exceptions to the copy-on-write rule: one is in the garbage collector itself, one in managing the root of the database, the others in the byte code interpreter built on top of Shades. For example, only occasionally do we have to copy the function's activation record when we update or initialize a local variable or a temporary value. We shall return to these exceptions in appropriate contexts. None of these mutations are apparent to Shines programmers.

1.2 Related Work

Copy-on-write–updates have been used previously, e.g. in shadow paging algorithms used for crash recovery [94, 31, 32, 46]. In all these cases, however, the functional updates have been hidden from the user by indirections, such as page tables, or by the execution mechanism of a less expressive database language. Indirections, particularly in a main memory environment, incur a slowdown which we chose not to take. Instead, we designed the Shines database programming language to accomodate Shades, not vice versa.

Relatively few complete database programming languages exclude destructive updates: relational, functional and deductive query languages are typically not computationally complete and most persistent programming languages, even some functional ones, include destructive updates [68]. STAPLE [60], FDL [84], Phil Trinder's thesis work [87] and the Glasgow transaction processor [4] are complete languages and exclude mutation from the programming model, but use mutation in their execution mechanisms, e.g. graph reduction for lazy evaluation, in manners unsuitable for Shades. Whether or not O_2FDL uses updates-in-place is unclear [56].

However, to see that our choice of forbidding mutation really is a good one, we have to compare our system to other persistent programming languages that have mutation.

Napier88 [61] has its background in PS-algol [9] developed at University of St. Andrews already since the early 80s. Napier88 has parametric polymorphism, first class functions, and support for reflective programming and system evolution [48]. Napier88 provides orthogonal persistence for data, code and threads.

Tycoon-2 is a direct successor of the languages TooL and Tycoon-1 [59] developed at University of Hamburg since 1989. Tycoon-2 is object-oriented, provides multiple inheritance, parametric polymorphism, rich subtyping rules, and first class functions. In addition to orthogonal persistence, Tycoon-2 provides also orthogonal mobility for data, code and threads [58, 57]. Tycoon's persistent threads were first presented as a database programming concept particularly well-suited for applications that manage long-term, distributed or cooperative activities [58]. Our system also has persistent threads and, as we shall see in Chapters to come, several other common features with Tycoon-2.

Several other persistent programming language projects have been conducted. Recently Java has also been regarded as an interesting candidate to be extended with orthogonal persistence [62].

Napier88, Tycoon-2, persistent Java as well as our Shines have numerous features in common: they are orthogonally persistent, they are all based on an a virtual machine, they are statically typed whenever possible, and they provide a library of index structures and threads with explicit concurrency control instead of transactions.

Unfortunately, while all these systems have convincing feature lists, they are almost entirely confined to the academia and have practically zero market penetration. Evidently there is considerable reluctance to adopt new technologies which require radical changes in languages, software development practice, or in overall system architecture [26], but additionally we speculate current systems are nowhere near relational DBMS in terms of developer support, demonstrated reliability, and performance. We will return in Chapter 6.4 to discuss how we hope to survive on the market.

Apparently no comparative performance data on persistent programming languages has ever been presented. We have no reason to believe their performance would be competitive to relational databases since even object oriented databases, whose data models are much simpler than the ones of persistent programming languages, are generally regarded to be considerably slower than serious relational databases. In the experimental part of this Thesis we aim to demonstrate that the performance of our system is superior to commercial relational main memory databases.

We shall return to related work relevant to persistent garbage collection, functional index structures, and byte code interpreters and compilers in appropriate places of Chapters 2, 3, and 4, respectively.

1.3 Introduction to the Thesis

Most database management systems have a layered structure. Assuming a coarse three-layer division of a traditional database manager, the lowest layer performs disk IO, provides media recovery, e.g. by mirroring or RAID, and crash recovery, e.g. with logs and periodic checkpointing. The second layer implements index structures, e.g.

B-trees, on top of the primitives provided by the lower level. The highest level builds a data model abstraction on top of the index structures, interprets the query language, e.g. SQL, and communicates with the clients of the database.

This Thesis follows the layered structure described above. In Chapter 2 we shall present Shades, the garbage collection and recovery algorithm, followed by a brief analysis and discussion on some less trivial details in our implementation of Shades.

Chapter 3 discusses the index structures implemented on top of Shades. Since mutation is prohibited, index structures differ from traditional textbook solutions. Furthermore, since we assume a main memory environment, efficient garbage collection and memory allocation, our cost model, the basis of data structure design, is very different from those in traditional database management systems where B-trees are ubiquitous.

As shown in Chapter 4, Shines is compiled to byte code and executed by a byte code interpreter. All data structures used by the byte code interpreter, including function activation records and byte code sequences, are allocated in the Shades database. Consequently also byte code threads are recoverable and resumed after crash recovery.

Chapter 5 defines and conducts a set of benchmarks and dissects the results. An abundance of further research topics will be put forth in Chapter 6. Chapter 7 concludes.

Chapter 2

Shades

Shades, our persistent memory manager, is unfortunately a rather large algorithm. Even without IO code and various auxiliary features our implementation of Shades requires approximately 3000 lines of carefully crafted and rather well commented C code. Trying to understand Shades from the C code, or even a pseudo-code version of it without any intermediate steps would be a daunting task. Instead, after defining the terminology, we approach Shades in steps: we start from a simple stop-and-copy collector, enhance it first to collect recently allocated data and then mature data, describe the principles of recovery, and finally show how to handle various persistent metadata.

2.1 Definitions and Terminology

In this Chapter, the word *application* refers to any software, e.g. index structures and interpreters, built on top of Shades.

A value is *persistent* (or *durable*, or *stable*) if it is backed up on disk and resilient to crashes due to, e.g., power failures. Otherwise the value is *transient* (or *volatile*).

The smallest unit of memory in Shades is the *cell*. It corresponds to a Pascal record or a C struct. The application constructs all other data structures — lists, strings, trees, tries, objects for object-oriented programming, byte code programs, stack frames — using cells. Cells are varisized: at least a few words, typically four to twenty words, or up to a few hundred words for very static data. Cells can contain e.g. integers, floating point numbers, and addresses of other cells.

The recovery algorithm of Shades requires that the size of a cell p can be determined by inspecting p itself and possibly cells referred to by p. Under the same condition, it must also be possible to determine whether or not the words in the cell p are addresses of other cells.

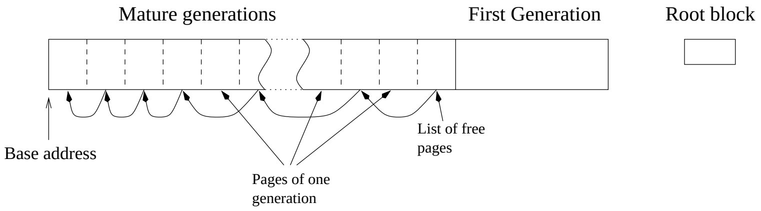


Figure 2.1: Main memory organization in the database image.

The *root block* is a small fixed-size array of pointers to cells and various administrative data. The root block is always updated in place. The pointers of the root block are regarded as the *root set* of the garbage collector; a cell is *live* if it is referred to by a pointer in the root block or in another live cell. Cells which are not reachable from the root set are *garbage* and Shades will eventually collect them away. For simplicity we may assume the root block itself does not contain cells.

When the application wishes to create a new cell, Shades allocates it from the *first generation*, a contiguous area of memory typically from a few hundred kilobytes to a few megabytes in size. Cells in the first generation are transient and they can be updated in place.

When the first generation becomes exhausted, or perhaps because of a timer signal, Shades copies all live cells in the first generation into a new *mature generation*, writes the new mature generation and the root block to disk. This is roughly the equivalent of performing a group commit. The root block is written atomically using duplexed writes [32] to specific positions on disk. After writing the root block and clearing the first generation, Shades may perform some additional garbage collection in the mature generations before returning to execute application code.

Cells in mature generations are persistent, but immutable to the application programmer. Consequently, no cell in an older generation can contain an address of a cell in a newer generation.

Cells in mature generations are located on *pages* for disk IO. Pages are of a fixed size and typically rather large, e.g. 64 kB. Each page has a counter which tells how many cells the page contains.

The *database image* consists of the root block, the first generation, possibly thousands of mature generations, and of free pages not belonging to a mature generation. Since dynamic memory allocation take place in the first and mature generations, they are often also collectively called the *heap* of the program. Figure 2.1 shows one convenient organization of the database image in main memory.

The *address* of a cell in the database is the offset of the cell from the beginning of the database. Therefore, although the base address of the database may change after recovery or vary from machine to machine, the addresses of the cells in the database remain unchanged. In pseudo-code and informal explanations we almost always refer to cells with their addresses; e.g. the phrase "the cell p" actually would stand for "the cell stored starting at the address p". As in most programming languages, there is also a special address NIL which does not contain a cell.

Finally, for sake of exactness, we define a *pointer* to be a memory word containing an address, i.e. the value of a pointer is an address. This definition differs slightly from typical C programmer parleance where the terms pointer and address are often used interchangeably. In Shades all recoverable pointers are either in the root block or in cells in mature generations.

The expression LOC(x) evaluates to the address of x, and its inverse operator LINK(p) evaluates to the value referred to by p. LOC corresponds to the C prefix operator `&` and LINK corresponds to `*`.

Additionally Shades contains numerous transient data structures reconstructed during recovery. The pool of free pages, the bitmap of disk allocation, etc. are all transient. Each generation and each page has a descriptor which will be discussed later. Usually these descriptors are stored in an array indexed by generation and page numbers respectively. A handful of programming tricks and transient data structures described in Section 2.7 find efficiently the page and the generation where any given cell resides.

2.2 Simple Stop-and-Copy Collection

Shades is developed from a stop-and-copy garbage collector, originally introduced by Fenichel and Yochelson [23] in 1969 and improved and varied by many others since. In stop-and-copy garbage collection the heap is divided into two semispaces, the Fromspace and Tospace. During garbage collection all live, or non-garbage, cells in the Fromspace are copied to the Tospace. The application continues allocation in the Tospace until it becomes full. Before garbage collection Fromspace and Tospace are flipped: the Tospace becomes the new Fromspace and vice versa.

The pseudo-code for a simple stop-and-copy collector is given below. Basically it is trivial recursive copying of cells. However, care must be taken when a cell is referred from several locations. In order to preserve the sharing of the cell, a *forwarding address* is stored in the original cell in the Fromspace to refer to the copied cell in the Tospace after the cell has been copied. Whenever SIMPLE-COPY visits a cell, it first checks whether it has already been copied, i.e. it is a forwarding address, and if so, then SIMPLE-COPY returns the address of the copied cell. Otherwise the cell is copied into Tospace and pointers in the copied cell are traversed recursively.

```
SIMPLE-COLLECT()
Flip the roles of Fromspace and Tospace
for r ∈ pointers in the root block do
    r ← SIMPLE-COPY(r)
```

```
SIMPLE-COPY(p : address of a cell)
if p = NIL then
   return NIL
else
   if LINK(p) is a forwarding address then
      return the forwarding address in LINK(p)
   else
      q ← allocate from Tospace the number of words in the cell LINK(p)
      Copy contents of the cell LINK(p) to the cell LINK(q)
      Write in LINK(p) a forwarding address to q
      for r ∈ pointers in the cell LINK(q) do
         r ← SIMPLE-COPY(r)
      return q
```

We shall return later, in Section 2.7, to low-level details of how to represent cells and forwarding addresses in our implementation of Shades.

A typical implementation of a semispace is a contiguous area of memory. In such cases memory allocation in a semispace can be implemented using an allocation pointer which is initiated to the beginning of the semispace when the semispace is cleared and incremented by the size of the cell whenever a cell is copied into or allocated from the semispace. Memory exhaustion can be tested by comparing the allocation pointer to the end of the semispace.

The above scheme is also used for the first generation in our implementation of Shades. However, the first generation is always used as a Fromspace, never as a Tospace. The Tospace is always a mature generation, which consists of pages which are not necessarily located adjacently in the database. Each page, however, is a contiguous piece of memory and the above allocator can be used within that page. When the page becomes full, a new empty page is taken from the pool of free pages and added to the generation serving as the Tospace. Additionally we have to maintain the counter of cells for each page. This allocator has an important property to be used in recovery: all cells of the generation can be enumerated in the order they were allocated in the generation.

In drawings we will omit pages, since they would easily clutter the pictures; instead we visualize generations as varisized contiguous areas of memory.

The simple stop-and-copy collector presented above does not satisfy one important criterion required by the recovery algorithm. It requires that cells which are referred to from outside the generation must be copied first into the Tospace generation. Cheney's [15] elegant non-recursive version of the stop-and-copy collector satisfies this criterion. However, in our implementation we found the below algorithm to be slightly more efficient, probably because it inspects each cell only once while Cheney's collector inspects them twice, once in the Fromspace and once in the Tospace.

The algorithm below can be used to collect the first generation in Shades under some very limited circumstances. It uses an auxiliary transient stack, the *copy_stack*, to store addresses of pointers to cells which have not yet been copied. The depth of the *copy_stack* is at most the maximum number of cells in the first generation. Since the first generation size is usually also an upper limit to the mature generation size, we can note that the *copy_stack* need not be very large. Actually *copy_stack* could be replaced by any implementation of an ordered multiset, but a stack is one of the simplest and most efficient.

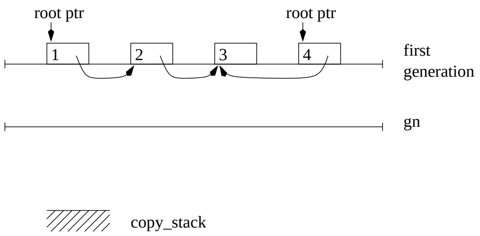
Stepping through the example in Figure 2.2 probably helps to understand the pseudo-code.

```
FG-Collect()
gn ← allocate a new mature generation
Clear the copy_stack
for r ∈ pointers in the root block do
  if r ≠ NIL then
    r ← FG-Copy(r)
FG-Drain-Copy-Stack()
Clear the first generation
Write the pages of gn to disk
Write the root block to disk
```

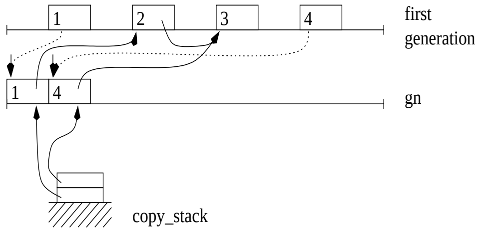
The variable *gn* does not contain the actual memory image of the generation, but a handle to the transient descriptor of the generation. As a trivial optimization we have also moved the test for NIL to an earlier place in execution.

```
FG-Drain-Copy-Stack()
while copy_stack is not empty do
  pp ← pop from copy_stack an address of a pointer
  p ← LINK(p)
  q ← FG-Copy(p)
  LINK(pp) ← q
```

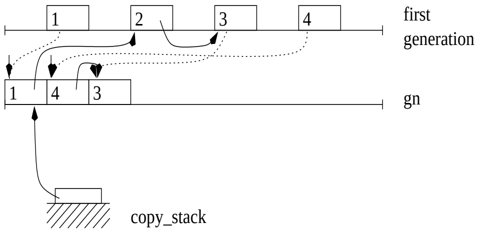
```
FG-Copy(p : address of a cell)
if p refers to a forwarding address then
  return the forwarding address in p
else
  q ← allocate from gn the number of words in the cell LINK(p)
  Copy contents of the cell LINK(p) to the cell LINK(q)
  Write in LINK(p) a forwarding address to q
  for r ∈ pointers in the cell LINK(q) do
    if r ≠ NIL then
      Push LOC(r) into copy_stack
  return q
```



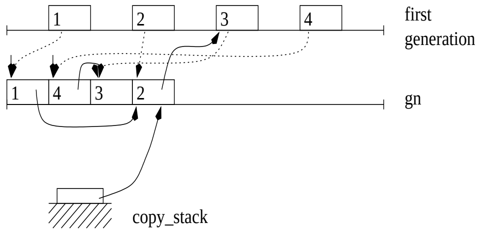
On line 3 in FG-Collect, initial state. Possible garbage cells are not drawn.



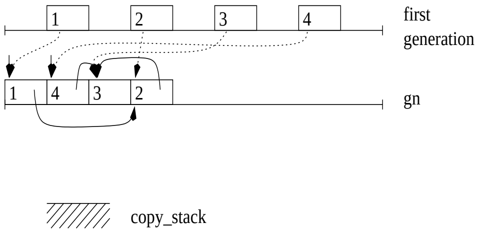
On line 6 in FG-Collect, after copying the cells 1 and 4 directly accessible from the root set. Pointers in the copied cells were pushed to the *copy_stack*. Dotted lines are forwarding addresses.



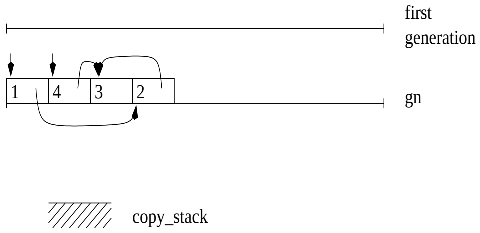
After one iteration in FG-Drain-Copy-Stack. The pointer from cell 4 to cell 3 was popped and cell 3 was copied.



After the second iteration in FG-Drain-Copy-Stack. The pointer from cell 1 to cell 2 was popped, cell 2 was copied, and the pointer from the copied cell 2 to cell 3 was pushed to the *copy_stack*.



After a third iteration in FG-Drain-Copy-Stack the *copy_stack* has become empty. The forwarding address of cell 3 was used.



After line 7 in FG-Collect the first generation has been cleared. As soon as pages of *gn* and the root block have been written to disk, allocation may resume from the first generation.

Figure 2.2: An example of FG-Collect in action.

2.3 Mature Generation Garbage Collection

The standard stop-and-copy collector has several appealling properties. For example, it is rather efficient, it handles varisized cells easily, and it essentially solves all memory fragmentation problems. Also memory allocation can be done efficiently.

On the other hand, having two semispaces consumes prohibitively much precious memory resources. Furthermore, each semispace must have a reasonable amount of free memory, otherwise the garbage collector will be called too frequently.

Additionally the stop-and-copy collector halts the execution of the application for the entire duration of the collection. For a 200 MB semispace this would require somewhere from five to ten seconds of CPU-work and tens of seconds, perhaps even a minute, of disk IO. However, for small heaps, say up to a megabyte, CPU consumption as well as disk IO time remains acceptable. Particularly for the first generation, where usually only a minority of cells survive the collection[1], stop-and-copy collection behaves well. But the cornerstone in all persistent garbage collection has been the treatment of mature generations.

In Shades, garbage collection of mature generations, also called *major collection*, advances from the youngest mature generation to the oldest. Only one or a few mature generations are collected at a time, then we return to execute application code, collect the first generation (*minor collection*), and write the root block before resuming to major collection for the next few mature generations. In other words, major collection, application processing, minor collections and writing root blocks are interleaved in order to give each task a fair amount of CPU and disk IO time.

Furthermore, in Shades only the mature generation currently being collected needs a Tospace. After the mature generation has been collected, the pages in the Fromspace mature generation are returned to the pool of free pages. Since generations are small compared to the whole database, only a negligible amount of memory is duplicated.

Should two mature generations be sufficiently small, the mature garbage collector may merge them. This keeps the number of mature generations from growing unlimited.

When a generation gn is collected, all younger, already collected generations must be regarded as a root set to gn. But scanning all generations younger than gn for pointers to cells in gn is very expensive; should gn be the oldest generation, we would have to scan almost the whole database. In other words, in order to collect n mature generations, we would have to scan $O(n^2)$ generations.

Fortunately, before arriving to collect gn, we have already collected all generations younger than gn and we could scan for pointers into cells in gn already then. This leads us to a technique called *remembered sets* [88], frequently abbreviated as *remsets*. In our variation of remsets each mature generation gn to be collected dur-

[1] See Section 5.2 for experimental evidence of this

ing the major collection is attached an ordered set of addresses of pointers to cells in gn. When we copy a cell and notice a pointer in it to a generation gn which has not yet been collected, the address of the pointer is added to the remset of gn. This includes first generation collections.

Remembered sets are transient and no disk IO is caused from inserting and removing addresses of pointers from remsets. In the next Section 2.4 we will see how remsets are reconstructred during recovery. The implementation details of remsets are important for overall performance and memory usage, and are discussed in Section 2.7. Although remsets can, at least theoretically, consume a considerable amount of memory, experiments described in Section 5.3 show that this rarely is a reason for concern.

Mature generations can be marked to be in various states: generations marked NORMAL have not been and will not be collected during this major collection, generations marked TO_BE_COLLECTED will be collected during this major collection, and the generations marked BEING_COLLECTED are currently being collected. After the generation has been collected, its remset and main memory pages can be freed. The transient metadata and the disk pages occupied by a collected generation may, however, not yet be freed. This will be further elaborated when we see how recovery works.

It is convenient to have the metadata for the mature generations in two lists: the *from_space* list is for generations marked TO_BE_COLLECTED, and the *to_space* list for new NORMAL generations. Since new data is inserted also to the end of *to_space*, it is convenient to have a pointer also to its last element. The list *to_space* contains all mature generations when major collection is not active. Both lists are ordered according to age from the youngest to the oldest generation.

```
MAJOR-COLLECTION-BEGIN()
1  Mark all generations TO_BE_COLLECTED
2  from_space ← to_space
3  to_space ← NIL
```

The above procedure performs the equivalent of flipping Fromspace and Tospace. It is called in the beginning a major collection. Scheduling major collection with the rest of application will be discussed in Sections 2.7 and 5.

Merging generations is done on the first line in MAJOR-COLLECTION-STEP by having *from_gns* contain several generations. Heuristics for choosing when to merge are discussed later.

```
Major-Collection-Step()
from_gns ← Take one or a few youngest generations from from_space
for from_gn ∈ from_gns do
    Mark from_gn as BEING_COLLECTED
to_gn ← allocate a new mature generation
Mark to_gn as NORMAL
Put to_gn last in to_space
Clear the copy_stack
for from_gn ∈ from_gns do
    for pp ∈ the remset of from_gn do
        p ← LINK(pp)
        q ← Copy(p)
        LINK(pp) ← q
for r ∈ pointers in the root block do
    if r ≠ NIL then
        r ← Copy(r)
Drain-Copy-Stack()
for from_gn ∈ from_gns do
    Free the pages of from_gn
    Free the remset of from_gn
    Retire-Generation(from_gn)
Write the pages of to_gn to disk
```

We shall return to the procedure Retire-Generation in the next Section. When *from_space* becomes empty, the major collection is finished and Major-Collection-End is called. It too is defined in the next Section.

Below we give a revised version of the minor collection of the first generation. Note that both major and minor collection use the same functions Drain-Copy-Stack and Copy. If no major collection is in progress, remsets are not in use and the now obsolete procedure FG-Copy works identically to Copy in collecting the first generation.

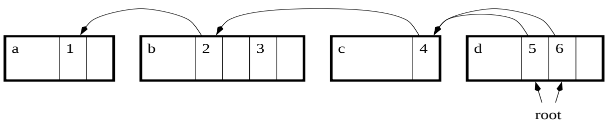
```
FG-Collect()
to_gn ← allocate a new mature generation
Mark to_gn as NORMAL
Put to_gn first in to_space
Clear the copy_stack
for r ∈ pointers in the root block do
    if r ≠ NIL then
        r ← Copy(r)
Drain-Copy-Stack()
Clear the first generation
Write the pages in to_gn to disk
Write the root block to disk
```

```
Drain-Copy-Stack()
while copy_stack is not empty do
    pp ← pop from copy_stack an address of a pointer
    p ← LINK(pp)
    q ← Copy(p)
    LINK(pp) ← q
```

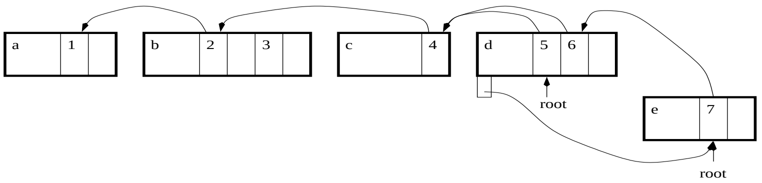
```
Copy(p : address of a cell)
if LINK(p) is a forwarding address then
    return the forwarding address in LINK(p)
else
    q ← allocate from to_gn the number of words in the cell LINK(p)
    Copy contents of the cell LINK(p) to the cell LINK(q)
    Write in LINK(p) a forwarding address to q
    for r ∈ pointers in the cell LINK(q) do
        if r ≠ NIL then
            gn ← the generation of the cell referred by r
            if gn is marked TO_BE_COLLECTED then
                Put LOC(r) into the remset of gn
            else
                if gn is marked BEING_COLLECTED
                ∨ Fromspace is the first generation then
                    Push LOC(r) into copy_stack
    return q
```

An example should clarify the algorithm. Assume the database looks as follows before initiating the major collection. Boxes marked with characters represent generations and boxes marked with numbers are cells in the generations. Generation a is the oldest generation and generation d is the youngest. Solid arrows are pointers. As mentioned earlier, we omit pages from the illustrations and assume generations are contiguous.

Generations typically contain thousands of cells, but for simplicity we are only going to look at a few cells in this example. Clearly cell 3 is garbage.

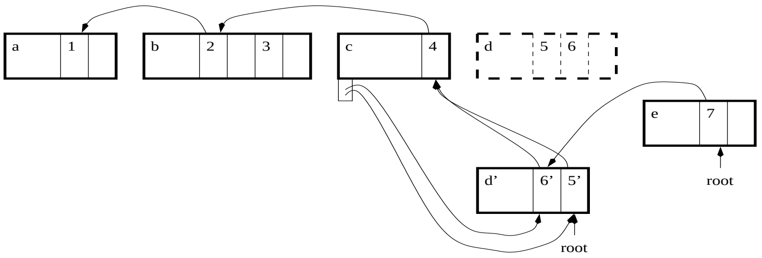


In the beginning of a major collection, all generations are marked TO_BE_COLLECTED and put into the *from_space* list by Mature-Collection-Begin. Assume a first generation is collected and it becomes the new youngest mature generation e, and assume e contains a cell 7 which refers to cell 6 in generation d. Since generation d is marked TO_BE_COLLECTED, the procedure Copy will insert the address of the pointer from cell 7 to cell 6 into the remset of generation d.



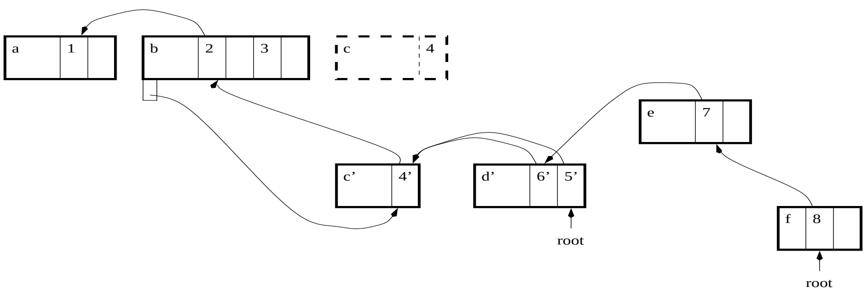
Now assume we collect the mature generation d. The Tospace mature generation of d is called d' and cells are marked correspondingly. Cell 6 is copied because it is referred to by a pointer whose address is in d's remset and cell 5 is copied because a root pointer refers to it. After copying cells from d into d', the pages occupied by d will be returned to the pool of free pages.

Note that the pointer from cell 7 to cell 6 is updated in place to refer to the copied cell 6'. While the application is not allowed to update values in the mature generation, Shades itself will do it. In the next Section 2.4 we shall see how this affects recovery.



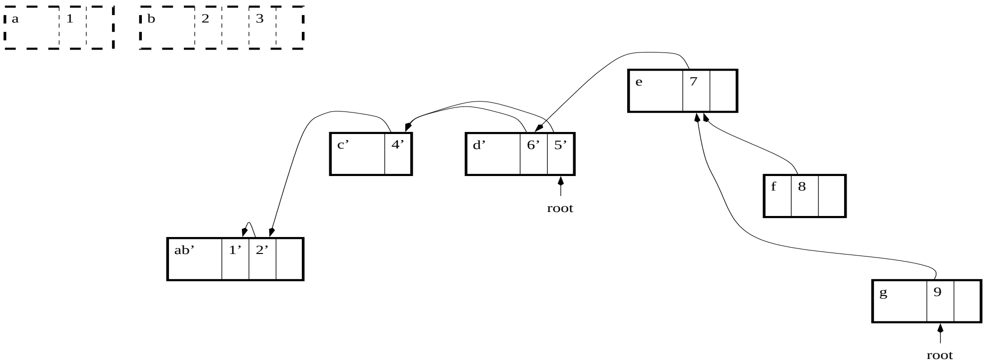
Assume there is a pressing need for more free pages, so we collect a second mature generation c to c' before returning control to the application. Here cell 4 was copied to cell 4' because addresses of pointers from 6' and 5' belonged to the remset of c. When handling the pointer from 5' to 4, the forwarding address from 4 to 4' was used. Naturally the address of the pointer from 4' to cell 2 will be added to the remset of generation b.

After the application has run for a while, a first generation collection occurs and a new mature generation f appears.



Since the mark of generation e is NORMAL, the remset of generation e is left empty even if cell 8 in f refers to cell 7 in e. Also the generations c' and d' are marked NORMAL and generation a is still marked TO_BE_COLLECTED.

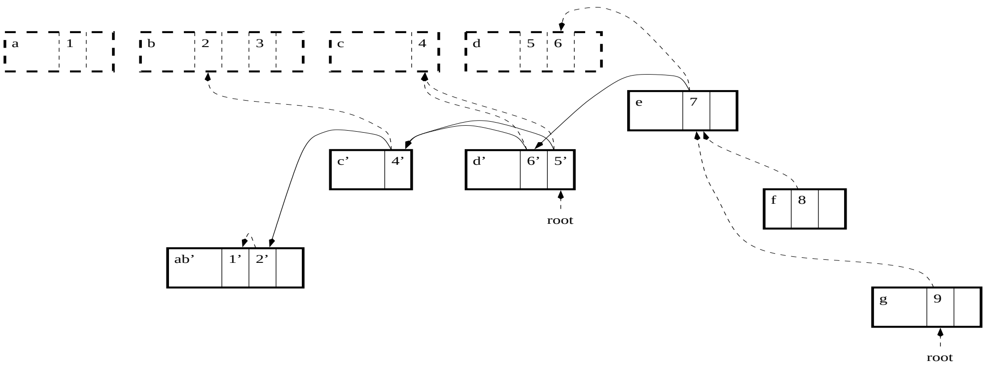
Next the collector wants to merge the generations a and b into ab'. There is little to this except that pointers from b to a can be treated without remsets, as if they were normal intra-generation pointers. The mature collection is now finished. In the final picture, a new mature generation g has appeared after some application processing.



2.4 Recovery

While the application is not allowed to mutate cells once they have been collected to mature generations and written to disk, Shades itself does exactly this on line 12 in Major-Collection-Step when traversing the remsets. The mutated pages will *not* be rewritten to disk and there will be no logs or similar structures on disk of these mutations, yet still we can reconstruct the new page during recovery.

We shall first take a look at how to recover the database when a major collection had been completed before the crash. The example from the previous Section serves this purpose. Dashed lines show pointer values on disk pages and solid lines how they were redirected in main memory after the pages were written to disk. We omit pointers from or within the generations a, b, c, and d since they shall not be used during recovery.



Recovery starts by reading from disk into main memory the pages of the generations in the original *from_space*, i.e. the generations a, b, c and d. These generations are marked TO_BE_COLLECTED and inserted into the *from_space*. We postpone until the next Section all issues related to metadata. For now we simply assume all transient metadata has already been reconstructed.

Next the pages of generation e are read from disk and scanned by calling Recovery-FG-Collect(e). Since there is a pointer from cell 7 to a generation marked TO_BE_COLLECTED, the address of that pointer will be added to the remset of generation d.

The generations d and e and the remset of d are now identical to what they were before collecting the generation d before the crash. Based on this premise, the call Recovery-Major-Collection-Step($\{d\}$, d') reads d' from disk and redirects the pointer from cell 7 to the cell 6'. It also scans d' for pointers to older generations and updates the remsets of older generations as if during normal major collection. Finally it frees the pages of generation d.

Unsurprisingly the next steps in recovery will be the recovery of the generations c', f, ab' and finally g. In short the whole sequence of recovery is

1. Read generations a, b, c, and d from disk, mark them TO_BE_COLLECTED and insert them into *from_space*.
2. Recovery-FG-Collect(e)
3. Recovery-Major-Collection-Step($\{d\}$, d')
4. Recovery-Major-Collection-Step($\{c\}$, c')
5. Recovery-FG-Collect(f)
6. Recovery-Major-Collection-Step($\{a,b\}$, ab')
7. Recovery-FG-Collect(g)

In abstract terms, recovery involves simulating the major collection in order to redo the pointer mutation performed by line 12 in Major-Collection-Step.

```
Recovery-Major-Collection-Step(from_gns, to_gn)
for from_gn ∈ from_gns do
    Remove from_gn from the list from_space
    Mark from_gn as BEING_COLLECTED
Read pages of to_gn from disk
Mark to_gn as NORMAL
Put to_gn last in to_space
for from_gn ∈ from_gns do
    for pp ∈ the remset of from_gn do
        p ← LINK(pp)
        q ← Recovery-Copy(p, to_gn)
        LINK(pp) ← q
Recovery-Scan(to_gn)
for from_gn ∈ from_gns do
    Free the pages of from_gn
    Free the remset of from_gn
    Retire-Generation(from_gn)
```

```
Recovery-FG-Collect(to_gn)
Read pages of to_gn from disk
Mark to_gn as NORMAL
Put to_gn first in to_space
Recovery-Scan(to_gn)
```

```
Recovery-Scan(to_gn)
for q ∈ all cells in to_gn in their order of allocation do
  for r ∈ pointers in the cell LINK(q) do
    if r ≠ NIL then
      gn ← the generation of the cell LINK(r)
      if gn is generation marked TO_BE_COLLECTED then
        Put LOC(r) into the remset of gn
```

In the procedure Copy the copied cell was always scanned immediately after its allocation from *to_gn*. Since the scan above proceeds in the same order, the remsets of older generations in *from_space* will be updated identically to the major collection before the crash.

The above procedure presumes the restriction given Section 2.1 that the pointers in the cell in q must be enumerated by looking into the cell q itself and possibly cells pointed to by the cell q, but no further: a cell referred by q might reside in *from_space* and we may be unable to follow a pointer value in it. Note also that a pointer in q may refer to a forwarding address which refers to a cell in *to_gn*.

```
Recovery-Copy(p : address of a cell, to_gn)
if LINK(p) is a forwarding address then
  return the forwarding address in LINK(p)
else
  q ← pseudo-allocate the next cell in to_gn
  Write in LINK(p) a forwarding address to q
  return q
```

The ith issue of pseudo-allocation should return the same address as the ith allocation before crash in Copy. Pseudo-allocation should not modify the contents of *to_gn* in any way. It can be easily implemented by simulating the behavior of the allocator described in Section 2.2. In order to advance the allocation pointer we need to know the size of the cell in q. Here the same restriction applies for determining the size of q as for enumerating pointers in q for function Recovery-Scan.

During recovery the pages of *from_space* serve only three purposes: as a store for forwarding addresses, for reading some metadata as described in the next Section, and for cells referred to by cells whose size or pointers are being determined. If we would require that the size and pointers of a cell depend only on data in the cell itself, then we could avoid reading from disk at least some of the generations a, b, c, and d. This would undoubtedly reduce the recovery times for Shades, but it would also make it slightly more cumbersome to implement, e.g., some features and optimizations in the byte code interpreter.

Consider now a database crash in the middle of a major collection. For example, if the database had crashed after committing generation f and unless b were a generation collected from a first generation, the pointers from generation b to a would not be reconstructed by the recovery steps described earlier. To overcome this problem, also generations a and b must serve as *to_gn* during some RECOVERY-MAJOR-COLLECTION-STEP. While some optimizations again exist, one simple alternative to ensure this happens is to require that each recovery must contain one full major collection in addition to the abrupted one. In other words, if the database crashed after writing f, then recovery would start from the Fromspace of generations a, b, c and d, and only after that proceed to using them as Fromspace to recover generations e, d', c' and f.

With the above scheme, generations and their pages on disk may be freed only after the completion of the major collection in which they were taken from the *from_space*. We retire former *from_space*-generations to the set *old_from_space*.

```
RETIRE-GENERATION(gn)
  old_from_space ← old_from_space ∪ {gn}
```

```
MAJOR-COLLECTION-END()
  for gn ∈ old_from_space do
     Mark disk pages of gn free
     Discard the transient metadata of gn
  old_from_space ← {}
```

During recovery we assumed that no pages were corrupted because of the crash. Numerous alternatives exist to achieve this, but the simplest is to use simple shadow paging: we write only to locations on disk which have been marked free in a past committed group. We almost always do this in any case because disk pages are freed only by the rather infrequent calls to MAJOR-COLLECTION-END. However, consider what could happen if all functions MAJOR-COLLECTION-STEP, MAJOR-COLLECTION-END and FG-COLLECT were called in the same commit group and FG-COLLECT would reuse some of the disk pages in *old_from_space* freed by the MAJOR-COLLECTION-END. If a crash occurred during writing those disk pages and corrupted the old data, then we would have neither finished the most recent major collection nor could we redo the previous major collection in *old_from_space*. A simple remedy for this problem is to refrain from calling MAJOR-COLLECTION-STEP in the same commit group with MAJOR-COLLECTION-END. Now the major collection will finish one commit group before reusing the disk pages.

2.5 Metadata

Until now we have ignored the details of how to manage and particularly how to recover the various metadata in Shades. For example, we have not described how to locate the disk page corresponding to a main memory page or how to recover all sufficient information of generations and their pages.

Numerous alternatives exist to solve these problems. In our system, the persistent metadata of the youngest generation is stored in the root block. After group commit this information is copied to the first generation. Mature collection steps store further information on the collected generations into the first generation.

Recall the example from the beginning of Section 2.4. The root block contains persistent metadata to read in generation g. It contains metadata on generations ab' and f. Since f was created by a collection, the pointers from g to f are up-to-date on disk and can be followed without RECOVERY-MAJOR-COLLECTION-STEPS. Generation f contains metadata on generations c', d' and e. A corresponding sequence of descriptions exist in generations a, b, c and d. Each metadata cell contains the page numbers and disk pages numbers for the generation.

These persistent metadata cells form linked lists. A pointer in the root block refers to the list of metadata for generations ab', f, c', d' and e. Generation e contains the metadata of generation d. A second pointer refers to the metadata for generations a, b, c and d, and a third pointer would refer to the metadata for generations collected in the previous major collection round. These lists contain the information to restore the transient lists *to_space*, *from_space* and *old_from_space*. Older and no longer referred metadata becomes garbage and is collected away by Shades itself.

On all workloads we have tested, less than 0.1% of disk writing is generation metadata. Only for pathologically small page sizes, e.g. a few hundred bytes, generation metadata consumes noticeable amounts of memory. On the other hand, page sizes less than 8 to 16 kB decrease disk IO considerably and a page size less than a few kilobytes limits the maximum cell size impractically low.

Each page contains a counter of the number of cells in that page. We choose to store this counter in the first word of the page so that no additional metadata cells are required. This also ensures the address zero (0) can serve as a NIL address.

Disk page management is done with a simple transient bitmap. Recovering the bitmap is easy: initially all disk pages are marked free, and each time a disk page is read into a memory page, the disk page is marked to be in use. Free lists for pages and generation identifiers are recovered similarly by reconstructing them after recovery.

Note that the memory pages read from disk for recovering metadata may be in use also in older generations. For example, generation g may reuse some of the pages used in generation c. Therefore it may be necessary to reread generation g later in recovery.

2.6 Analysis

In his excellent survey of uniprocessor garbage collection [91], Paul Wilson states "standard textbook analyses of garbage collection algorithms usually miss the most important characteristics of collectors — namely the constant factors associated with the various costs, such as write barrier overhead and locality effects." Therefore we have dedicated the entire Chapter 5 to studying experimentally Shades as well as the index structures and the byte code interpreter. Some crude analysis can, however, be done mathematically.

The asymptotic complexity of a stop-and-copy collection is proportional to the amount of copied data R. It does, at least not theoretically, depend on the size of the database M. The garbage collector reclaims all garbage $M - R$. Assuming the application allocates S bytes, then $\lceil S/(M - R) \rceil$ collections will have to be issued. Since each collection costs $\mathcal{O}(R)$ time, we can write the overhead of stop-and-copy as $\mathcal{O}(R/(M - R))$ per unit of allocation. Note that — at least theoretically — we can reach arbitrarily small overheads by increasing M sufficiently. Conversely, if $M = R$ no allocation can be done at all.

In practice, however, we can do better than $\mathcal{O}(R/(M - R))$ since garbage is usually not evenly distributed in the database and we can try to collect generations which can be assumed to have the highest density of garbage. This will be discussed in Sections 5.2 and 6. On the other hand, various caching and paging effects have a decreasing impact on performance as M increases. Generational stop-and-copy garbage collection and memory reference locality has been widely studied in literature [33, 18, 28, 27, 74, 75, 92, 93, 95].

One full major collection must be performed during recovery. Immediately before finishing the next major collection almost three full images of the database must be kept on disk. After the next major collection finishes, disk space consumption drops back to twice the database to gradually increase again.

Setting an upper bound for the recovery time seems to be a little harder since it depends on many implementation details, such as major garbage collection scheduling. If we can ignore the cost of reading in metadata, the worst case is the equivalent of reading $3M$ bytes from disk in a situation discussed above for disk space consumption. However, in a simplistic implementation we may have to read the entire generation to access the small metadata cell in it, and since those generations may be overwritten in memory during processing previous major collections, we may have to set a safe upper bound to as high as $5M$. In practice, however, the average recovery times in our system range from the equivalent of reading $2M$ to $3M$ bytes from disk. The disk reads are long and easily parallellizable resulting in recovery times often much shorter than booting the machine.

The worst case in remset size is when as many pointers as possible in the database refer to the same cell. In such a case, remsets can consume as much memory as the entire database. Naturally this is a highly pathological case; such a data set would be utterly meaningless. Furthermore, such cases can be avoided by extending Shades to duplicate the popular cell into for example ten generations of different age and have pointers refer to the newest possible cell. In practice, as discussed in Chapter 5, we expect remset sizes to range from negligible to at most a tenth of M.

If major collection scheduling is successful, then Shades is real-time in terms of disk writing per group commit. However, the CPU work needed by the loop traversing the remset on line 9 in MAJOR-COLLECTION-STEP depends on the remset size, which may be considerable, e.g. in the pathological case in the previous paragraph. Duplicating popular cells may alleviate this problem, but interlacing the remset traversal loop with minor collections solves it entirely. This would require having a separate *copy_stack* for minor collections. Furthermore, since the application may want to access cells in the Fromspace, we have to store forwarding addresses elsewhere.

Since Shines does not have pointer-wise comparison, neither duplicating the popular cells nor interlacing remset traversal with minor collections is apparent to the Shines programmer.

2.7 Implementation

As mentioned earlier, we have implemented the Shades garbage collector and recovery algorithm ourselves. The implementation was never aimed to result in a flakey, stripped-down research prototype. We have aimed to produce well-designed, solid, clean, and efficient code. Code reviews, profiling, and heavy testing with aggressive fail-fast programming techniques have been used intensively to this end. To date we have recovered a deliberately crashed database for over 100.000 times without a single failure. The code and database images have been tested to work on and across numerous UNIX-like platforms, including 64-bit processors and different byte orders.

The interface from Shades to the higher layers is delightfully simple:

1. For each cell there must be code that computes the size of the cell and enumerates all pointers in it.
2. A root block is essentially a global record of words and pointers.
3. A function which tells whether the first generation contains the given number of unallocated words.
4. A function which allocates a cell of the given size.
5. A function which commences group commit, clears the first generation, and possibly performs some garbage collection in the mature part of the database.

In addition to a run-time for a persistent programming language, the persistent garbage collector and its index structures can be used also from C and with smart pointer interfaces also from C++ and Java. Unfortunately the C interface will have to take into account the invalidation of pointer values outside of the root block during group commit. Furthermore the Java interface is sluggish, partly because of Java itself and partly because the smart pointers are removed from the root set only after the smart pointer objects have been finalized by the Java garbage collector. Nevertheless, our implementation of Shades was designed to be flexible enough to facilitate the implementation of dedicated database engines, possibly with new application oriented cell types and index structures.

2.7.1 Cells

Since it must be possible to add new kinds of cells without breaking existing modularity, we have reserved the topmost eight bits from the first word of each cell as a type tag for the cell. The codes that determine the size and enumerate the pointers of the cells are indexed with these type tags. Approximately 50 different cell types are currently reserved for various purposes, leaving an abundance of cell types for future use.[2] The lowest 24 bits of the first word of each cell can be used by the application.

Forwarding addresses are their own cell type. The actual forwarding address is stored in the second word in the cell. Since any cell may be written over by a forwarding address, no cell may be shorter than two words.

Currently cells can consist of raw data words, pointers, non-nil pointers, tagged words [34, 6, 89], and also of unused words. Non-nil pointers are a slight optimization and also a basis for some assertions during debugging. The meaning of tagged words depend of their two lowermost bits, giving 30 bits for raw data or significant pointer bits. Since cells are word-aligned, thirty bits will suffice in 32-bit machines.

2.7.2 Memory Organization

Contrary to pages in mature generations, the first generation allocator allocates downwards. This has the benefit of leaving the allocated cell's address conveniently in the allocation pointer, which we try to allocate globally in a CPU register using compiler-specific tricks [85]. In addition to testing memory exhaustion, a pointer to the end of the last page can also be used to test whether a cell belongs to the first generation. The resulting allocator is therefore identical to the one used in Appel's landmark work on SML/NJ [5, 6].

Memory allocation can not fail. Before allocation the caller must check whether there is sufficient free memory in the first generation. This can be done with simple

[2] Note that the user-defined types in persistent programming language discussed briefly in Chapter 4 do not consume the cell type tag space.

pointer arithmetic. Furthermore, since a single check can cover several eventual allocation requests, this explicit check incurs a negligible overhead and hardly motivates the inaccessible memory region trick suggested by Appel [5].

If possible, also the base address of the database is allocated globally in a CPU register.

Pages are located adjacently and starting from the base address. Furthermore, since we require the page size to be a power of two, a simple shift-right instruction of the address of a cell gives the page number of the cell. This idea has been used also in train collectors for translating cell addresses to train numbers [37, 80]. A transient array indexed by the page number gives the generation of the page.

Remembered sets are implemented as linked lists, but so that several consecutive values are combined to a single linked list node in order to decrease memory overhead. Unused remset nodes are organized in a freelist.

When debugging, all cells in the first generation are interleaved with a one word "red zone". The consistencies of these red zones are checked occasionally. This, and some other similar tricks have helped us to detect and find several programming errors in Shades, index structures, byte code interpreter, and even in Shines code.

2.7.3 Major Collection Scheduling

The longer we can wait before we start a mature generation collection, the more data has become garbage by the time garbage collection starts, and the more efficient garbage collection will be. But, should only little garbage be collected from the newest mature generations, the major collector will have to perform numerous steps to make room for next minor collections, resulting thereby in poor realtimeness.

As we are writing this, we do not know how to solve the above problem for all possible workloads. Instead we give the database administrators three parameters to tweak: *start_gc_limit*, *mid_gc_limit*, and *mid_gc_effort*. Figure 2.3 shows how much major collection effort, measured in terms of disk writing per major collection step, will at least be done given a certain amount of free memory in the database.

Since the youngest mature generation is the most recently collected first generation and no space would be released when collecting it immediately again, it is beneficial to wait for one or a few minor collections after MAJOR-COLLECTION-BEGIN before issuing the first MAJOR-COLLECTION-STEP. Furthermore, as shown in Section 2.4, we have to refrain from issuing MAJOR-COLLECTION-END in the same commit group in which we issued MAJOR-COLLECTION-STEP. The minimum time in which the whole database can be collected is therefore two commit groups. Therefore, as long as the amount of free memory in the database is less than the equivalent of two first generations, maximum major collection speed will be used.

In our current implementation of Shades the database will crash if even the maximum major collection effort fails to provide sufficient free memory. The database

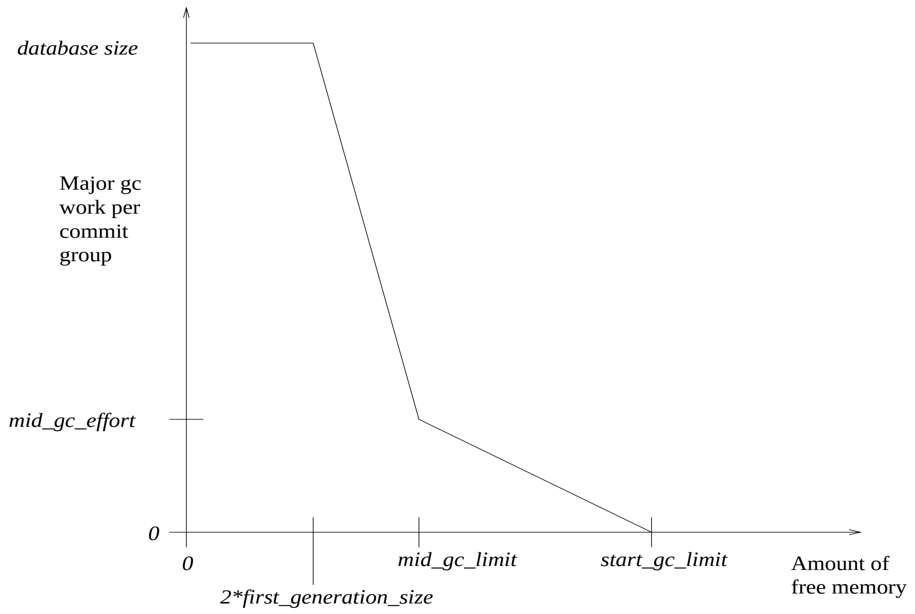


Figure 2.3: Control parameters and dependence of major collection effort per commit group on the amount of free memory.

image can, however, be recovered and execution can be continued if more memory has been allocated for the database. In practice this solution is not satisfactory and will have to be improved in future: the database manager might kill low-priority transactions and run user-defined procedures which discard data of less importance. Weak pointers [91], or pointers whose weakness depends on the amount of free memory, may also provide useful for memory conservation.

For reasons explained in Section 5.3, whenever mature garbage collection is active, we issue at least one MAJOR-COLLECTION-STEP even if the amount of free memory would exceed *start_gc_limit*.

2.7.4 The IO Layer

When updates are frequent, disk writing typically becomes the performance bottleneck in main memory databases. Fortunately disk writes can be made more efficient by writing in parallel to several disks, i.e. by employing *disk striping*. In a single-threaded system parallel writing can be conveniently achieved using asynchronous writes: while one disk performs asynchronously the previous write, the processor can already issue the next write to another disk.

In Shades there are some additional opportunities for asynchronous writing. In particular, since mature garbage collection is done in the beginning of commit groups, writing collected mature generations to disk can be done concurrently with serving new transactions.

Currently asynchronous writes have been implemented using both POSIX.4 `aio_write`, Solaris `aiowrite`, and with normal UNIX `write`s issued concurrently using POSIX.4a threads. All pending asynchronous writes must be finished before writing the root block. The benefits of asynchronous writes will be discussed in Section 5.4.

Parallel disk reads conversely speed up recovery. Currently parallel reads are used to read all the pages of a generation. It might be possible to read also several generations in parallel, but this would require changes in the recovery algorithm.

In addition to actual disk IO and disk space management, the IO layer also performs possible byte order translations when reading blocks during recovery.

As currently implemented, our database server includes no support for media recovery. It would, however, be trivial to implement e.g. simple mirroring by modifying only the IO code.

The interfaces and behavior of the IO layer have been studied in greater detail by Antti-Pekka Liedes [51].

2.7.5 Concurrency

In disk-based databases transactions frequently request information not resident in main memory. Since disk reads are rather slow, in the order of 10 milliseconds, and because other transactions could be served and other hardware resources utilized during that 10 ms period, most disk-based database servers are multi-threaded. Typically each transaction is assigned to a thread which performs the index operations, disk reads and log writes according to the transaction's query plan. In order to keep shared resources, such as the buffer cache and various data structures, consistent, threads protect them using locks, or *latches*, as the database world calls them.

Unfortunately efficient programming with threads and locks is notoriously hard, particularly if the threads share intensively common data structures protected by locks. Merely successfully taking and releasing a lock typically lasts in the order of one microsecond, and if the lock was already taken, from ten or twenty up to several hundred microseconds pass before another thread advances. Current computers can easily perform hundreds of simple arithmetic operations in one microsecond. Furthermore, pre-emptively multithreading programs tend to be very hard to debug, profile and test.

In a main memory database no disk reading is necessary after recovery. In our database server disk writes and network activities can occur only at a few places in code and execution. Given the horrors of multi-threaded programming described above and the simplified setting main memory database servers, we chose to make our database single-threaded. All memory management, index operations and query execution is performed by the same thread, without any locking and overhead caused thereof.

Naturally, in order to serve several transactions concurrently, the query interpreter will have to implement its own threads. This work, however, turned out to be a very small matter of programming once the architecture and run-time organization was implemented as described in Section 4.2.

Although we need no concurrency control to keep data structures structurally consistent, we do need concurrency control to keep data structures transaction-consistent. At this point it is not clear how we will choose to do this, but numerous approaches have been discussed in literature.

2.8 Related Work

As far as we know, no garbage collector except ours has required the application to treat the contents of mature generations as immutable. We shall call mutable mature generations *partitions* in accordance with the literature discussing such garbage collectors. Consider the changes needed if Shades did allow mutation. We would run into at least three considerable sources of overhead.

First, these destructive updates would have to be logged on disk. No separate checkpointing would, however, be needed as the checkpointing pass over the database can be merged with the major collection of all partitions.

Second, we would also have to maintain remembered sets of pointers from older partitions to newer ones and to update them during pointer mutations. Also the first generation would need a remset. In case the pointer values are mutated again, these remembered sets would have to be pruned away their outdated pointer values during collection. Unlike remsets for pointers from newer to older partitions, which can be discarded as soon as the older generation has been collected, the remsets for pointers from older to newer generations will have to remain intact and occupy precious memory resources over major collection rounds as well.

Third, we would have to reconstruct the old-to-young partition remsets somehow during recovery. A simple scan over the older generations will not suffice since the relative order of pointers in the remsets is also significant. One solution might be to represent old-to-young generation remsets in some sorted order, but this would naturally be computationally expensive.

Alternatively to reconstructing remembered sets during recovery, we could keep them in stable storage. Updating pointers in a cell and especially moving a cell to another place will require changes in the remsets and therefore considerable increase in disk activity.

Most partitioned garbage collectors are optimized for heaps and remsets larger than the main memory. In such cases considerable care must be taken to avoid unnecessary disk reading and writing of remsets. These collectors usually assume only the collected partition and its remset and other metadata must be present in main memory during collection.

PMOS [63, 64] stores remsets attached to each partition. These remsets also contain the address of the referred cell, not only the address of the referring pointer. When a cell is moved by the collector and the referring pointer is not in main memory, the change in the cell's location is stored in persistent lists to be patched into the referring pointer the next time when it is read into main memory. Maheswari's algorithm [53] maintains both the incoming and outgoing references of a partition in stable store. The information is kept in a more compact form than separate simple lists.

In PMOS the recognition of garbage cells is based on train collection [36]. Maheswari's algorithm reclaims groups of objects spanning over several partitions using a global marking phase, which is piggybacked with intra-partition collections to reduce disk overhead [53]. Other approaches include reference counting [8], marking a "cut" of the database [83], stop-and-copy collection [65, 50], and ideas shared with distributed garbage collection [52].

PMOS has its background in persistent programming languages and Maheshwari's algorithm in object databases. While PMOS actively reclusters objects, the latter expects the user to supply a good partitioning: their benchmarks assume from 0% to at most 15% inter-partition pointers, which seems rather optimistic given that a partition contains less than 0.4% of the objects in the database.

Furthermore, neither PMOS nor Maheshwari's algorithm promise to reclaim all garbage in a single collection. Although all garbage will be reclaimed eventually, it may require numerous passes over the whole database.

The algorithms of Kolodner [49] and O'Toole [65] are based on stop-and-copy collection with two semispaces. Kolodner's algorithm is based on the Ellis–Appel–Li collector [21] and uses a read barrier to arguably provide real-timeness. Kolodner's algorithm logs all changes done to the heap by the collectors, for example copying a single cell results in a log record containing the old and new addresses of the cell and the value written over by the forwarding pointer. This naturally results in large amounts of disk writing, although admittedly those log writes need not be forced.

O'Toole [65] replicates the heap so that the application can access one replica while the collector performs a standard stop-and-copy on the other replica. New cells and updates in the application's replica are logged. Their benchmarks report 80 transactions per second in a TPC-B benchmark similar to the one described in Section 5, which is two to three orders of magnitude less than our results. We suspect that they failed to use commit grouping, which would partly explain such poor results.

To our knowledge Shades is the only stop-and-copy collector which does not need two semispaces, does not use logs, and maintains remembered sets in transient memory.

While excellent surveys exist on transient garbage collection [91, 42] and distributed garbage collection [1], no survey of similar extent has been done on persistent garbage collection. In our opinion the best and most up-to-date survey is given in [53].

Chapter 3

Index Structures

Numerous data structures and algorithms have been presented in standard textbook literature, but practically all of them are *imperative*: they assume updates in place are possible anywhere in the data structure at any time. Recently researchers have become interested in *functional* data structures [66, 67]. This research can be applied to practice in purely functional programming languages, such as Miranda and Haskell, where no primitives for update-in-place are given to the programmer.

However, although Miranda and Haskell disallow mutation from the application programmer, all known efficient execution methods of these *lazy functional languages* frequently mutate old data structures [70]. Fortunately *strict functional languages* can be efficiently executed without mutating old values, making them amenable to Shades. We call data structures implemented in lazy and strict pure functional programming languages *lazy* and *strict functional data structures*, respectively.[1]

Interestingly, theoretical investigations [71, 11] have shown that there are problems which require linear time when implemented in a imperative or lazy functional language, but $\Omega(n \log n)$ time when implemented in a strict functional language. Whether a similar difference exists between lazy and imperative languages is unknown. Nevertheless, although an algorithm implemented on top of Shades may unavoidably be a factor of $O(\log n)$ slower than its imperative equivalent, we note that these cases seem to be rare in practise, and even then the apparently smaller constant factors in Shades may outweigh the increase in asymptotic complexity.

At this point we have to remind that only a part of the work on index structures was done by the author of this Thesis; a large portion of the theoretical and experimental work has been conducted by other people in the HiBase-project — either in

[1] An unfortunate disorder reigns in the terminology: numerous sources, including [66], frequently fail to clearly state whether an algorithm requires lazy evaluation, and [79, 20, 45] use the word "persistent" to refer to data structures whose old versions are retained during updates, but whose implementation may nevertheless require imperative update-in-place modifications. All functional data structures are automatically persistent in this sense. The concept of "Pure LISP" in some older articles is equivalent to a strict functional programming language.

the supervision of the author or despite of it. Proper credits are given to these kind souls at appropriate points.

3.1 Tree-like Data Structures

An imperative implementation of tree-like data structures can usually be rather easily translated to its functional counterpart: if a leaf node is modified, the path from the leaf to the root is copied yielding a new root. Sarnak and Tarjan [79] called this general technique *path copying* when implementing persistent data structures.

Path copying can usually be done without a slowdown in asymptotic complexity: since traversing from the root of a tree to a leaf node already requires $O(\log n)$ steps, rewriting the traversed path from the possibly modified leaf node back to the root does not increase the asymptotic complexity. Furthermore, since creating cells is very cheap in Shades, usually much cheaper than reading cells from main memory when traversing the path downwards, also the increase in constant factors remains small.

Consequently this section on trees will present mostly a brief introduction to the implementation and some low-level optimizations, but in Sections 3.2 and 3.3 we shall see how rejecting mutation affects data structure design more deeply.

3.1.1 Balanced Trees

Three different balanced trees have been implemented on top of Shades: Sirkku Paajanen [67] implemented leaf-oriented AVL-trees and internal node 2-3-4–trees and found the latter ones slightly slower for small trees. But since 2-3-4–trees contain larger nodes and relatively fewer pointers, they consume less memory, which in turn improves performance when the trees are very large and more effort is spent on garbage collection [67, p. 55]. Furthermore, internal nodes in leaf-oriented trees frequently contain references to keys already removed from the database, thereby preventing the garbage collector from reclaiming them.

Paajanen's work did not cover all necessary and conceivable tree operations. Therefore, in 1998, Jarkko Lavinen implemented a third set of trees, internal node AVL-trees, now complete with group and range operations and cursor-like functionality for scanning and updating keys and values. Some of the algorithmic choices were affected by Lauri Malmi's work [55].

Contrary to Paajanen, Lavinen used Shines instead of C, which considerably reduced the effort of implementation. Despite the overhead of the byte code interpreter, similar performance was reached when these measures were taken:

- To represent an AVL-tree node using a standard Shines tuple requires six words: the first word is reserved by Shades and the tuple size (as described in Chapter 4) and the following words are occupied by the AVL-balance, the key and value of the node, and addresses of the left and right subtrees. To conserve memory we chose to amend Shines with a new primitive data type, the AVL-node, which is represented by a single three to five word cell in Shades: the 24 lowermost bits of the first words are not reserved by Shades and can therefore be used to store the heights of the left and right subtrees, and the addresses to subtrees are stored in the cell only if they really are present. This reduces the memory consumption of AVL-nodes by a third.

- Seven new byte code instructions were written to query the configuration and fields of the AVL-node cells, and one new byte code instruction to create a new AVL-node cell when given the key, value, and left and right subtrees. Should the height difference between the left and right subtree be two, this instruction would also perform the necessary single and double rotations to restore the balance.
- A Just-In-Time compiler described briefly in Section 4.3.2.7 translates the most time-consuming functions, including AVL-routines, into C, spawns the C-compiler and dynamically links the object files into the database server.

3.1.2 Tries

If the key of the tree is, or can conveniently be converted to a sequence of bits, tries [24, 29] are usually a more efficient index structure alternative to balanced trees. At least ten different tries with varying compression methods, bucket and node sizes have been implemented and experimented on Shades.

Currently the most significant and most featureful trie implementation is due to Sirkku Paajanen and yours truly [67]. This trie has a node size of four. It uses *path compression* to compress sequences of single-child nodes into one node and *width compression* to remove null pointers from trie nodes. This trie is used to implement e.g. integer-keyed maps in Shines.

Iivonen and Tikkanen [39] have recently designed the *PWL-trie* which uses *level compression* in addition to the above compression methods. This trie combines two levels of uncombined nodes into one larger node whenever the combined node will have at most 13 children. For most key distributions this reduces the memory consumed by pointers and cell headers and results in memory savings up to almost 25% for various test inputs [39]. Neither of these two tries use bucketing.

Some of the tries implemented on Shades are dedicated to specific kinds of keys. Analysis trees, e.g., assume keys are telephone digit strings. Similar dedicated index structures may be implemented for string keys and IP routing tables.

3.1.3 Strings

In C, character strings are usually represented by arrays of characters, usually terminated by the nil-character. This gives excellent character lookup and update performance, but string concatenation takes linear time.

In Shades we chose to represent strings with tree-like structures, designed and implemented by Lars Wirzenius. The leaves of the tree contain a varying number of characters, typically from one to 32. Internal nodes of the tree behave somewhat similarly to nodes in B-trees except that keys are actually relative character positions from the beginning of the subtree. The root of the tree contains an offset into the string and its length. These two fields allow the programmer to skip characters from the beginning and the end of the string without having to copy internal and leave nodes of the tree, only the root node.

With this string representation character lookups and updates and string concatenation can be done in $O(\log n)$ time. Removing characters from the beginning and the end of strings takes constant time. Great care was taken to make it as efficient as possible to copy Shades' strings into C-representation and back. Long strings created from their equivalents in C consume approximately 30% more memory in Shades, but since parts of strings can be shared regardless of how the strings are manipulated, memory consumption may eventually be smaller in Shades than in C.

The set of string operations in Shines will be extensive, ranging from basic primitives described above to various conversion, searching and regular expression matching operations.

3.2 Double-Ended Queues

While tree-like data structures seem very amenable for functional programming, queues, double-ended queues (or *deques*) and mergeable priority queues with deletion are considerably more challenging. In an imperative language a deque might be implemented with a doubly linked list or a ring buffer should the number of items in the deque have an upper bound. Unfortunately arbitrary doubly linked lists can not be constructed and definitely not modified in strict functional programming languages.

An old trick to implement a strict functional queue is to divide the queue in two lists, the head and tail lists. The tail is stored in reverse order, i.e. the last item inserted to the queue is the first item in the tail. Items are removed from the queue by popping them away the head. Should the head be empty, the tail is reversed to a new head-part and cleared. On average this queue has constant insert and remove complexity, but the worst case complexity of a remove can nevertheless be $O(n)$ where n is the number of items in the queue.

In 1981 Hood and Melville [35] presented a strict functional queue which incrementally reverses the tail-part during other operations, thereby reaching constant time performance in all cases. In 1993 Chuang and Goldberg [16] extended the same idea to cover deques. In 1995 Kaplan and Tarjan [45] devised a technique called *recursive slow-down* and used it to design deques with constant-time catenation.

Four different queues or deques have been implemented on Shades. Paajanen implemented Hood-Melville–queues and designed *logarithmic queues* [67], where insertions to either end of the queue could be done in constant time, but deleting the first item from the queue required $O(\log n)$ time. In practice, however, the logarithmic queue was over two times more efficient and consumed 25–30% less memory than the Hood-Melville queue [67, p. 37]. This suggests that constant factors have a significant importance and may easily outweigh logarithmic factors in practice.

Unfortunately the logarithmic queue is not double-ended. Therefore two deques were designed aiming at minimizing the constant factors while retaining at most logarithmic worst case performance. Iivonen used buffers of at most k items in the head and tail of the deque and a tree-like structure to implement the deque of full buffers in the middle [38]. The value of k was determined experimentally and usually set to five. This structure has logarithmic average case complexity for all operations, but still outperformed Hood-Melville–queues in most tests. It also consumes almost 30–40% less memory than the logarithmic queue.

The author of this Thesis took the idea of buffering further and replaced the tree in Iivonen's deque with a deque of full buffers. In this respect it follows some ideas presented in [45], but with several simplifications and optimizations. This idea provided similar practical performance to logarithmic queues and almost identical memory consumption to Iivonen's solution. This deque is currently widely used, e.g. to implement scheduling and message queues between byte code threads, and probably merits a closer look.

A deque of items of type T can be one of the following:

1. NIL, signifying an empty deque.
2. A buffer containing from one to k items, where k may be, e.g., five.
3. A triple $\langle head, middle, tail\rangle$, where *head* and *tail* are buffers containing items of type T and *middle* is a deque containg full buffers of items of type T.

In other words, following the *middle*-pointers the deque appears like a stack of two increasingly deeper trees, where the tree nodes are implemented by buffers and all non-root tree nodes are full. Figure 3.1 shows one possible deque containing the integer items from one to 116. Rectangles represent buffers.

Many operations manipulate only the single buffer or the *head* and *tail* buffers of the topmost level. Occasionally, on average every fifth operation, we have to move full buffers from and to deeper levels. For example, when inserting an item to the end of the deque, if the *tail* buffer is full, the buffer is inserted last into the deque

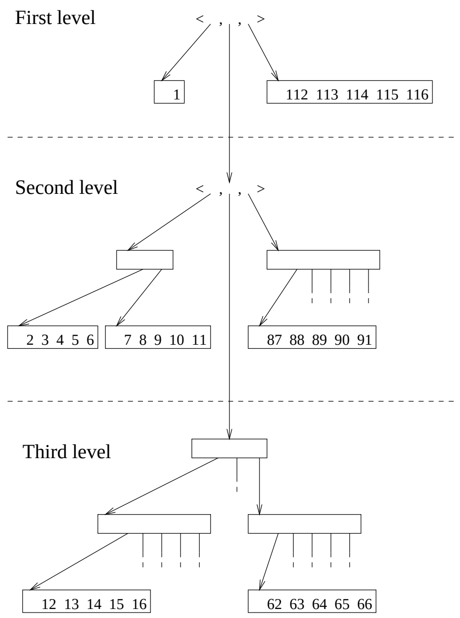


Figure 3.1: A deque of integer items one to 116.

middle and a singleton buffer with the new item becomes the *tail*. If *tail* is not full, the new item is simply inserted last into *tail* buffer.

In the code below we assume we have various insertion, access, and deletion routines for buffers readily implemented. In Shades buffers are implemented using single cells.

INSERT-LAST(*deque*, *item*)
if *deque* = NIL **then**
  return a buffer containing only *item*
elseif *deque* is a full buffer **then**
  return ⟨*deque*, NIL, a buffer containing only *item*⟩
elseif *deque* is a non-full buffer **then**
  return *deque* with *item* inserted last
elseif *deque* is a triple ⟨*head*, *middle*, *tail*⟩ and *tail* is a full buffer **then**
  return ⟨*head*, INSERT-LAST(*middle*, *tail*), a buffer containing only *item*⟩
elseif *deque* is a triple ⟨*head*, *middle*, *tail*⟩ and *tail* is a non-full buffer **then**
  return ⟨*head*, *middle*, *tail* with *item* inserted last⟩

Figure 3.2 illustrates the deque with the value 117 inserted last. In this case we had to insert a full *tail*-buffer the *middle* deque on both topmost levels.

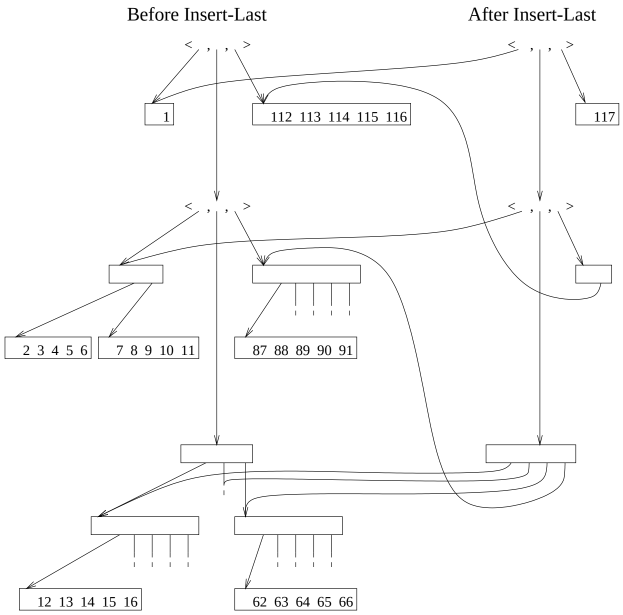


Figure 3.2: The deque before and after inserting 117 last.

Note that the *head* and *tail* in the triple are never empty. Therefore the first element of the deque is either the first element of the buffer or the first element of the *head* buffer in the triple.

When removing the first item in a deque with a *head* with only one item in it, the *head* will have to be refilled from the *middle*. This is illustrated in Figure 3.3. Should the *middle* be NIL, the triple degenerates to a single buffer.

REMOVE-FIRST(*deque*)

if *deque* is a buffer containing only one item **then**
  return NIL
elseif *deque* is a buffer containing several items **then**
  return *deque* with the first item removed
elseif *deque* is a triple $\langle head, Nil, tail\rangle$ and *head* contains only one item **then**
  return *tail*
elseif *deque* is a triple $\langle head, middle, tail\rangle$ and *head* contains only one item **then**
  return $\langle$first item in *middle*, REMOVE-FIRST(*middle*), *tail*$\rangle$
elseif *deque* is a triple $\langle head, middle, tail\rangle$ **then**
  return $\langle$*head* with the first item removed, *middle*, *tail*$\rangle$

Inserting a new first item and removing the last item are analogous to the previous two functions.

As mentioned earlier, only full buffers are pushed downwards to a deeper level. Therefore a singleton buffer on level i contains at least k^{i-1} items, i.e. the depth of

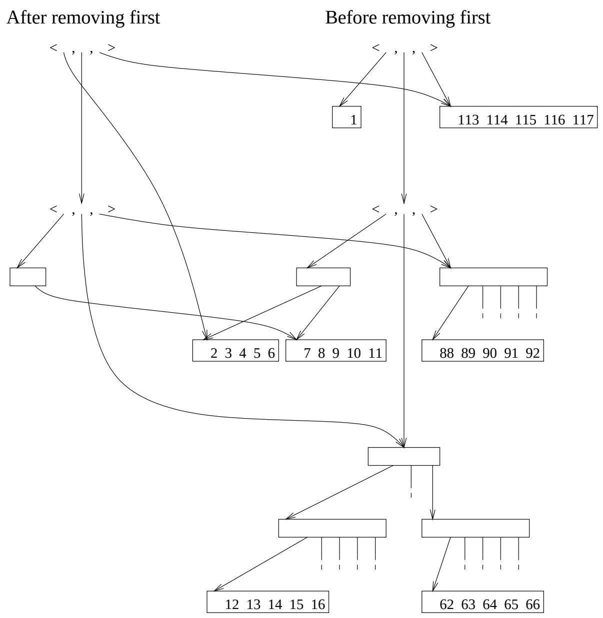


Figure 3.3: The deque after and before removing the first item.

the deque stack is $O(\log n)$. Since each call to an insert or delete spends a constant time on a level, the worst case cost of an insert or delete is logarithmic.

On average, however, this deque performs better, since we have to issue a recursive call only for full buffers in inserts and singleton buffers in deletions. Therefore a recursive call has a probability of 1/k. If we denote the cost of handling each level in the stack by c, an upper bound for the expected cost of an update is

$$\sum_{i=1}^{\lceil \log n \rceil} (1/k)^{i-1}\ c$$

Even if we let $n \to \infty$, we have a convergent geometric series for $k > 1$. Therefore, although our worst case performance is logarithmic, the average insert and delete cost is constant. In particular, we reach this average case performance always if we use the deque as a queue.

Actually, following the idea of Kaplan and Tarjan [45], it is possible to divide the cost of removes and inserts in deeper levels over all insert and remove operations so that a constant worst case performance can be reached. This approach, however, while theoretically interesting, unfortunately also complicates code and increases the constant factors so that we chose not to use it [13].

Note that it is possible to perform lookups and updates of the ith item in $O(\log i)$ time in all our deque and $O(\log n)$ all other queues except those based on lists, e.g. Hood-Melville–queues, where lookups take linear time. It is, however not possible to remove and insert new items anywhere except the edges of the deque, nor is it possible to catenate two deques or split a deque into two in constant time.

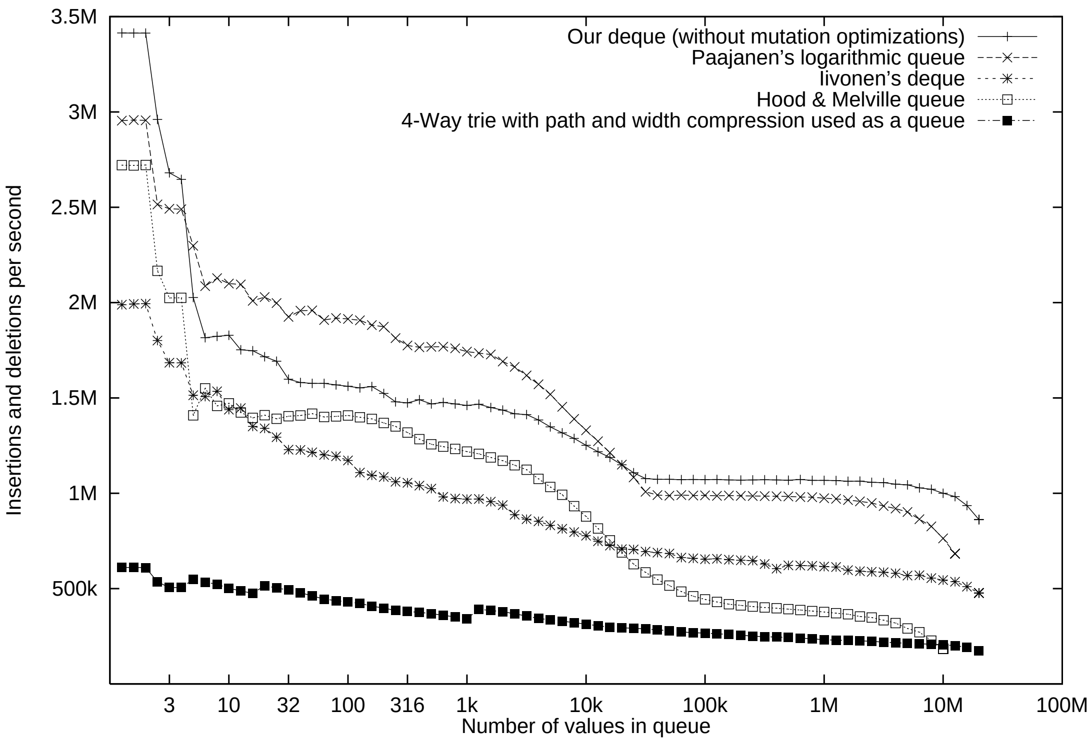


Figure 3.4: Queue performance benchmark. See Section 5.4 for the benchmark setting. The database was not backed up on disk.

3.3 Priority Queues

Priority queues, frequently also called *heaps*, provide at least the following operations:

1. Creating an empty priority queue and asking whether a given priority queue is empty.
2. Inserting an item into a priority queue.
3. Finding and deleting the item with the highest priority in the priority queue.
4. Merging two priority queues.

Additionally we may wish to be able to

5. Delete a given item from the priority queue.

but as we shall see, this will prove to be difficult in a strict functional language.

Numerous tree-like data structures have been proposed to implement priority queues. Various *heap-ordered* trees are one of the most common: here the item at each node has a higher priority than the items in its descendants. For example, Brodal and Okasaki [13] present an optimal solution that requires $O(\log n)$ time to delete the highest priority item, but uses only constant time for any other operation 1.–4. See also [66, 67, 47] for other implementations.

Deleting an arbitrary item from a heap-ordered tree is trivial path copying *providing we know the path from the item to the root*. In imperative implementations we can add a pointer from each item to its parent and eventually to the root. In addition to being forbidden in functional programming, this also prohibits storing the same item in several priority queues.

Apparently only two solutions have been presented in the literature [13]. The first one is to use two priority queues, one "positive" priority queue containing the actual items and another "negative" priority queue containing items deleted from the first priority queue. Deleting an item would insert it in the negative priority queue. Positive and negative occurrences of the same item cancel each other when deleting the highest priority item. Unfortunately deleting the highest priority item now has the worst case performance of $O(m \log m)$ where m is the minimum of insert and delete operations ever performed to the priority queue. Another problem is that items logically deleted from the priority still remain referred and therefore this priority queue "leaks memory" despite garbage collection.

The second solution is to use e.g. a binary search tree to represent the priority queue, possibly amended with a pointer from the root of the tree to the leftmost (or rightmost) item in order to give constant search time for the highest priority item. Both deletions and insertions would cost a logarithmic time. Unfortunately operation 4, merging two priority queues, would cost linear time. It might, however, be possible to perform the merge of the search trees incrementally during later insertions and deletions in the priority queue, but currently we do not quite know how to do it efficiently.

Neither solution is adequate although the latter solution has not yet been thoroughly investigated. A third solution might be to simulate an imperative priority queue: whereas imperative priority queues are given the address of an item to insert or delete, their functional simulations would be given an integer handle allocated by the caller and a function which compares the priorities of two handles. Pointers causing cycles in the imperative priority queue would be mapped through an indirection in the simulation. If we implement the indirection with a trie, we can perform reasonably efficient incremental merging. Naturally most updates in the simulation would carry an overhead of a logarithmic factor compared to the imperative version.

Clearly priority queues with deletion is an open theoretical problem, and therefore despite some trials [67], we have not yet decided our priority queue implementation(s) on Shades.

3.4 Low-Level Optimizations

Although by default cells in Shades are manipulated copy-on-write, when a cell resides in the first generation and it is not yet visible outside the program fragment that allocated it, the cell can be mutated in place. These optimizations are used carefully in only a few places in index structures and the byte code interpreter:

- Objects in Shines are implemented using tiny tries. The compiler can frequently deduce that an object is being referred to by only one variable in the program. Such objects can be updated-in-place. In benchmarks, although over a third of field updates were destructive, the speedup was hardly measurable.

- As discussed in Chapter 4.2, message queues, thread scheduling queues, context cells, and stack frames are updated in place whenever possible in the byte code interpreter. Updating stack frames in place reduces CPU-consumption on average by a third over a large set of benchmarks. The other optimizations reduce the cost of context switching and inter-thread messaging in the virtual machine. Combined they save another third on the very communication-intensive benchmark "Client" described in Section 5.1, but hardly nothing on other benchmarks.

Clearly there are places where update-in-place optimizations result in a worthwhile speedup. But these places are rare, sometimes hard to recognize, and according to our experiences, usually very hard to implement correctly so that they retain referential transparency. It also appears that tree-like data structures benefit very little from updates-in-place.

There are also some routines to retract all allocations since a given point in time in same commit group. These can be used to avoid useless first generation consumption when, e.g., an index operation that has already allocated some memory notices the operation will fail or be nilpotent.

Chapter 4

Shines

The design and implementation of the database programming language, Shines, built on top of Shades, is still underway. Numerous syntactic and semantic changes, most notably a module system and static typing, are imminent. Therefore, and because other texts describing Shines are in the process of being written, little attention will be paid to the Shines language from the prospective programmer's point of view; mostly we shall discuss the data structures, algorithms, design and implementation in the byte code interpreter and the back end of the byte code compiler.

Nevertheless, regardless of pending work, current Shines, also colloquially called Nolang, can already bootstrap itself and it has been used to implement a graphical user interface library and numerous benchmark and demonstration programs.

4.1 Shines Values as Shades Cells

Before going on to describe the byte code interpreter and compiler, we shall collect and review the methods we use to represent Shines values of various types using cells.

- As mentioned in Section 2.7.1, cells can contain tagged words [34, 6, 89]. The meaning of tagged words depend of their lowermost bits.

  A common solution for tagged words is to have the lowermost bits `00` signify a pointer and for example `01` signify an integer value. Following pointers can be done without further processing, but integer operations require a some bit operations in addition to the actual computation.

  In Shades following a pointer requires adding the pointer with the base address of the database to get the true address of the cell. Therefore we chose to use the opposite meanings of lowermost bits: now arithmetic operations on tagged integers need fewer bit operations and clearing the lowermost bits when following a pointer can be done by subtracting one from the base address of the database. The lowermost bit patterns `10` and `11` are reserved for future use.

- As discussed in Section 3.1.3, strings are represented by a tree-like data structure.

  String constants and function, type and field names are *interned* as in LISP-systems: each different string is given a unique integer which can be used for example for more efficient searching in various data structures.

- Lists, i.e. cons-cells, are their own cell type. The size of a cons cell is three words while many LISP-systems use only two words. On the other hand Shines has many other primitive data types in addition to lists and they are therefore probably less frequent in Shines than in LISP. Furthermore the 24 lowermost bits in the first word of cons cells are unused and might in future be used to store for example length information. This would be possible since the type system of Shines will guarantee that all lists will be properly NIL-terminated.

- n-tuples are represented as single cells of n tagged words in addition to the first word. The value of n is stored in the lowermost 24 bits of the first word.

  We have tentatively discussed adding tuples with named fields into Shines. These would differ from full-fledged objects in that they can not be multiply inherited.

- The current implementation of Shines does not yet have floating point values. A simple, though naturally suboptimal solution would be to "box" each floating point value into its own cell. Many, if not most, other implementations of programming languages do this.

- Neither does the current implementation of Shines support arbitrary precision integers, or bignums.

- As mentioned in Section 3.1.2, tries are used to implement maps with integer keys. While these can be used as normal arrays, they are efficient also when keys are distributed sparsely. The lowermost bits are shifted away from the tagged words representing the integer keys; otherwise the trie might be one level higher and all leaf nodes singleton nodes.

  Maps have their own syntactic sugar in Shines. For example the expression `m.[k]` finds the value of key `k` in the map `m`. The expression `m.[k := v]` returns a new map with the key `k` and its value `v` inserted (or replaced) into the map `m`. It is possible that other index structures, most notably the AVL-trees described in Section 3.1.1, currently used through explicit function calls will eventually be sweetened with similar syntac sugar.

- Objects are represented by a small header cell and a small trie. The 24 lowest bits in the first word of the header contain a number that represents the dynamic type of the object. The second word of the header refers to the root of the trie containing the fields and their values. The interned field name of the record is used as the key in the trie.

Most imperative languages implement objects using contiguous pieces of memory; we use a trie. A single-cell representation would be faster for lookups than a trie, but at least for large objects with many fields it would also be slower to update. But even in very object-intensive applications these costs remain reasonable: approximately 4% of CPU-time is used to read the field values and another 4% to update field values in the compiler. A trie representation also solves easily the problems related to schema evolution, first class fields, and indexing fields in multiply inherited objects.

Naturally, like maps, also objects have their own syntactic sugar. For example the expression `o.f` evaluates to the value of field `f` of object `o`. The expression `o.<f := v>` evaluates to a new object whose field `f` has the value of the expression `v`, but is otherwise identical to the object evaluated by the expression `o`.

- Lambda-functions, closures, and continuations [7] are represented with by stack frame cells, which will be discussed in Section 4.2.2.

4.2 The Byte Code Interpreter

Executing a byte code instruction consists of the following steps:

- Accessing the arguments of the instruction.
- Performing the function of the instruction.
- Storing the results of the instruction somewhere.
- Adjusting the program counter `pc` to refer to the next instruction and dispatching to it.

The mechanisms of instruction dispatching are discussed in Section 4.2.4.3 and also in conjunction with the byte code assembler in Section 4.3.2.6 and Just-In-Time compiling in Section 4.3.2.7. In the next Sections we shall present increasingly wider views of the virtual machine, which dictates how the three first steps should behave: first in Section 4.2.1 we study individual instructions, in Section 4.2.2 we extend the virtual machine with function closures and calls, and in Section 4.2.3 we discuss threads, inter-thread messaging and thread scheduling. Numerous subtle optimizations have been collected to Section 4.2.4 and, should the optimizations be related to the byte code compiler or the byte code assembler, also into Section 4.3.2.

4.2.1 Byte Code Instructions

Most byte code interpreters, including Postscript$^{\text{TM}}$ and the Java$^{\text{TM}}$ virtual machine, are based on some sort of a stack machine. This is a beneficial choice, since stacks of values are suitable for expression evaluation and they are an easy target language for the compiler. Furthermore, as discussed in [22, Section 2.3], stack machine interpreters can be more efficient than register machine interpreters.

A common solution to reduce the overhead of accessing the arguments and storing the result is to "cache" the topmost values of the stack in registers. This optimization is known in numerous forms, but we have chosen a simple and common one, used also in e.g. Objective Caml: we keep the last computed value in a virtual machine register called `accu`. If the last instruction is executed only for its side effects, e.g. printing on the screen, and it does not yield a meaningful value, `accu` remains unchanged. Specifically, `accu` is not automatically filled by popping a value from the stack, but the compiler will have to emit instructions for reloading the `accu` if it so pleases. The register `accu` also contains the return value and the last argument of function calls. In terms of Ertl [22], this is *static stack caching* with a one register cache and having it filled in the canonical state. This reduces argument access overhead by approximately 50% [22, Fig. 26]. Further, although gradually diminishing improvements would be possible by increasing the number of registers in the stack cache. But this would complicate the compiler, multiply the number of instructions possibly thrashing the instruction cache, and cause a shortage of registers in CISC processors such as the i386. Furthermore, the Just-In-Time compiler discussed in Section 4.3.2.7 has, in addition to many other benefits, similar effects to aggressive stack caching.

The stack grows upwards, towards higher addresses. The virtual machine register `sp` is the stack pointer. It holds the address of the first free slot in the stack. Therefore the C expression `*sp++ = accu` can be used to push the value of `accu` on the stack and `sp[-1]` can be used to read the topmost value on the stack. For example the addition instruction `add` is implemented with the C expression `accu += *--sp`.

To reduce the overhead of dispatching, we have merged frequently consecutively executed instructions into one. For example, picking values from stack and pushing them immediately back on stack are frequently performed several times before issuing function calls. We have therefore written such amusing instructions as `pick_push_pick_push_pick`, which takes three values from given positions in the stack and pushes them back on top of the stack.

The program counter `pc` was mentioned earlier. It refers to the beginning of the instruction being executed. Immediate arguments can be read `pc`-relatively. For example loading a given constant 42 to `accu` is represented by the byte code instruction `load_imm +42` and executed by the C expression `accu = pc[1]`.

In order to reduce dispatching time, all instructions performing arithmetic have a counterpart with an immediate constant. For example, to add the constant 42 to `accu` can be done with the single byte code instruction `add_imm +42`, implemented as `accu += pc[1]`, instead of performing the three instructions `push`, `load_imm +42`, and `add`. Furthermore, the compiler takes into account the commutativity of addition and generates the single `add_imm` instruction for the addition in both expressions $x + 42$ and $42 + x$.

Each instruction must modify the `pc`, otherwise the interpreter enters an endless loop. Increasing `pc` by the size of the current instruction proceeds to the successive instruction while adjusting it otherwise performs a branch. For example, the C code

```
if (accu == 0)
  pc += 2;
else
  pc += pc[1];
```

performs a branch to the label given as an immediate argument if `accu` is non-zero. The byte code assembler, described briefly in Section 4.3.2.6, translates symbolic labels into offsets from the current instruction.

Naturally we have merged comparison and conditional branch instructions to avoid the cost of dispatching between them. This has also the benefit of leaving a possibly meaningful value in `accu` instead of setting `accu` to a temporary boolean value which would be a known constant in the then- and else-branches. This optimization is described further in Section 4.3.2.2.

In Shines, as in other functional languages, local variables are created by `LET`-expressions. The expression `LET` *var* `:=` *val* `IN` *expr* evaluates the expression *val*, binds it to *var* whose scope is the expression *expr*. The value of the `LET`-expression is the value of *expr*. In the byte code interpreter local variables are simply values in the stack: we evaluate *val* to `accu`, push it on the stack, proceed to evaluate *expr*, and finally, unless optimized away, we decrease `sp` to discard the value of *val* from the stack.

Finally, for better or for worse, we have to admit that Shines does have global variables which appear to the application programmer as if they could be mutated in place. In reality, however, these global variables are stored in a trie referred to by the virtual machine register `globals`. Reading the value of a global variable performs a search in the trie using the interned string id of the name of the global variable as the key. Updating the value of the variable is a corresponding strict functional update in the trie.

4.2.2 Byte Code Blocks and Calls

Byte code instructions are stored in *byte code blocks*. Semantically a byte code block corresponds to an *extended basic block* [3, pp. 714] in the compiler: the byte code block can only be entered from its first instruction, it may have multiple exits, and it has no backward branches. Specifically, byte code blocks are executed atomically in terms of garbage collection and byte code thread scheduling. A byte code block is either suspended before completing its first byte code instruction or alternatively executed until it exits.

Byte code blocks are implemented as single cells in Shades. In addition to the byte code instructions, byte code block cells contain various auxiliary information used for example in profiling and debugging support. They also describe the state of the stack of a byte code thread about to execute the byte code block. This information is used by Shades to collect the stacks of byte code threads. Beginnings of byte code blocks are therefore the *gc-safe* points in Shines [42, 91].

A digression on terminology: although byte code instructions are in fact represented by entire words in Shades as well as in many other systems, the term byte code has persisted to be used for all kinds of virtual machine instructions stored in an array or a list.

In conventional programming languages stacks are implemented as large contiguous memory areas. In Shades this could be expensive due to the copy-on-write principle. Additionally traditional contiguous stacks are somewhat clumsy for implementing advanced control structures, most notably continuations [7, 82]. Therefore instead of storing the entire stack in a single cell we chose to have a cell for each individual stack frame. In addition to arguments and the execution stack of the function call, each stack frame contains a pointer `bcode` to the byte code block being executed and a pointer `next` to the stack frame to which it returns when the function returns. The virtual machine register `frames` refers to the first stack frame in the linked list created by the `next`-pointers.

The mechanisms of calling and returning are probably best illustrated by an example. Consider the three functions

```
foo(x,y) := x+bar(y+1);
bar(x) := zap(x+2);
zap(x) := x+3;
```

The function `foo` is compiled into two byte code blocks, one (`foo.1`) for computing `y+1` and calling `bar`, another (`foo.2`) for adding `x` to `accu` and returning. The function `bar` is compiled to one byte code block (`bar.1`), which adds 2 to `accu` and issues a tail call to `zap`. Function `zap` is likewise compiled to only one byte code block, `zap.1`, which adds 3 to `accu` and returns. Function `foo` requires one word of stack space to store the value of `x`, functions `bar` and `zap` require none since all other computations happen entirely in `accu`.[1]

For all byte code blocks representing function entry points there is a *prototype stack frame*. This is stored in the `globals` trie described in the previous Section. Issuing a call to a function is done by *taking a copy* of its prototype stack frame, setting the `next`-pointer of the copied stack frame, pushing all but the last argument to the copied stack frame and leaving the last argument to `accu`, and setting `frames` to refer to the copied stack frame. Figure 4.1 shows the stack frame when we are about to enter function `foo`.

[1]Should the compiler be allowed to inline the calls to `bar` and `zap`, only three instructions would result from the function `foo`: `add_imm +6`, `add`, and `return`.

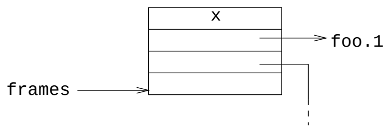


Figure 4.1: Stack frames before entering function `foo`. The register `accu` holds the value of `y`.

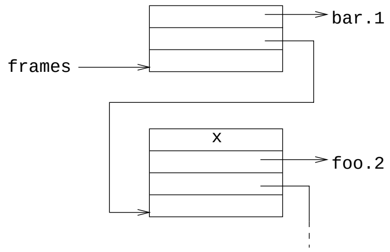


Figure 4.2: Stack frames before entering function `bar`.

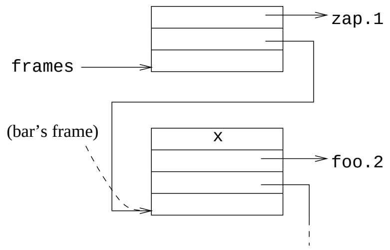


Figure 4.3: Stack frames before entering function `zap`.

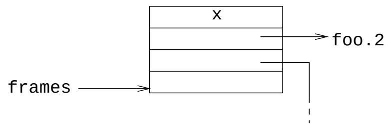


Figure 4.4: Stack frames after returning to function `foo`. The register `accu` holds the return value of `zap` and `bar`.

Before executing the instructions in the byte code block the interpreter checks whether the first generation contains sufficient free memory to serve the maximum amount of memory that executing the byte code block may require. If the first generation is about to exhaust, the virtual machine registers are stored in the root block and Shades is asked to collect the first generation and possibly perform some mature generation garbage collection. Since Shades changes pointer values, the virtual machine registers have to be reread from the root block before retrying to execute the byte code block. Rereading virtual machine registers from the root block is also the point from where execution resumes after recovery.

The byte code instructions in `foo.1` increment `accu` and issue a call to `bar`. Since we wish to return to `foo`, we set the `next`-pointer of the callee's stack frame refer to the caller's stack frame. Furthermore, since we do not wish to return to the byte code block `foo.1` but rather to `foo.2`, we change the `bcode`-pointer in the caller's stack frame to refer to `foo.2`. This can be done in-place, since the execution of `foo.1` appears atomic to Shades and the stack frame was already copied before executing `foo.1`. The stack is now illustrated in Figure 4.2.

The instructions in `bar.1` increment `accu` by 2 and issue a *tail* call to function `zap`. As in Scheme, tail recursion must be executed in constant space also in Shines. In other words, stack frames of functions that have issued a tail call must become garbage. This is accomplished simply by not retaining the stack frame of tail-caller in the list of frames, i.e. the callee's `next`-pointer is set to the same value as the caller's `next`-pointer. Figure 4.3 shows the stack frames after issuing the call to `zap`, but before entering it. Note that `bar`'s stack frame still refers to `foo`'s stack frame, but nothing refers to `bar`'s stack frame.

Returning from a function is to set `frames` to the value of the `next`-pointer, as shown in Figure 4.4. It may, however, be necessary to copy the stack frame before executing the byte code block. Whether or not copying is necessary is discussed in Section 4.2.4.1.

Shines, as many high-level programming languages, has *first class functions*: it is possible to create new functions on-the-fly, treat them as ordinary data, and call them later. For example, consider the function `make_adder` defined below. When given an argument `x`, it creates a new function which, when called, returns the sum of its argument and `x`.

```
make_adder(x) :=
  LET f(y) := x+y
  IN f;
```

First class functions usually contain *free variables*, which are neither arguments of the function nor defined by `LET`-expressions in it. In the above example the variable `x` is a free variable in `f`. The values of free variables must be stored in a *closure*.

We have chosen to represent closures using stack frames and free variables are regarded as additional arguments to the function. In the above example the compiler lifts the local unary function `f` into a global binary function `f'(x,y) := x+y`. When we call `make_adder`, it takes the prototype stack frame of `f'`, pushes the value of `x` on the copied stack frame, and returns the frame. Calling closures is essentially identical to calling global functions: the closure stack frame is copied (since it may be called several times), the `next`-pointer is set, all but the last argument are pushed, the last argument is left in `accu`, and `frames` is set to refer to the copied stack frame. In practice, however, as a speed optimization we have implemented dedicated byte code instructions for calling global functions.

Closures could have been organized in numerous alternative ways [7, pp. 112–114, 142–144]. Some representations, particularly the ones based on sharing cells, exclude the live-precise garbage collection optimization discussed in Section 4.3.2.3. Our representation, on the other hand, excludes mutation of the local variables — but this is consistent with the general design of Shines.

Recursive local functions need to access their own closure in order to issue the recursive call. Mutually recursive local functions require cyclic closures which refer to each other. This is possible in Shades if we allocate both closures in the same byte code block. We have, however, at least temporarily opted for a solution where the programmer is required to implement recursive local functions using the *Y-combinator* [70]. For example, the factorial function could be implemented as a local function as

```
LET fact(n, f) := IF n <= 1 THEN 1 ELSE n*f(n-1, f) IN expr
```

and called with `fact(n, fact)` in *expr*. Various syntactic sugar is being designed to dress recursion and particularly Y-combinators in an easier notation. Some of this syntac sugar has been described in Section 4.3.2.1.

We shall conclude this section by discussing *continuations*, the most powerful control structure in Shines. Continuations are snapshots of the computational state of the current thread. These snapshots can be taken, stored as ordinary data, and resumed at a desired moment. Continuations can be used to implement numerous other control structures, including conditionals, iteration, exceptions, coroutines, and logic variables [82].

The instruction `call_with_current_cont` issues a call to a given function f and passes as argument the current value c of `frames`. The continuation c can be resumed, or "returned to" by using the instruction `return_to_cont`. It stores the value of c in `frames` and proceeds as we would have returned from a function call. This has the effect of returning to the caller of f. It is also possible to return to a continuation several times, for example in order to implement retryable exceptions. The "return value" of f is the content of `accu` when `return_to_cont` is issued. It is also possible to return from f the normal way, by following the `next`-pointer.

4.2.3 Threads and Messaging

The values of the virtual machine registers `pc` and `sp` are initialized based on information in the byte code block before executing it. The other virtual machine registers `accu`, `frames` and `globals` constitute the *context* of the thread together with to the registers `messages`, `priority` and some other information to be described later. If the thread is currently executing all virtual machine registers are stored in C variables. Some effort has been taken to ensure the compiler allocates at least the most frequently used registers `accu`, `sp` and `pc` in processor registers. The virtual machine registers of threads currently not running are stored in context cells.

Each thread in the system is identified by a unique integer, the *thread id*. A pointer in the root block refers to a `threads` trie, which contains all the context cells in the system. The thread id is used as the search key in the trie.

The root block also contains pointers to four *scheduling deques*, each containing thread ids of threads waiting to be executed. Each scheduling deque represents one priority level: a thread can be executed only if there are no higher priority threads and threads on the same level are time-shared in a round-robin manner. While our scheduler is still rather rudimentary, it is nevertheless similar in spirit to real-time operating systems, e.g. QNX.

Resuming a thread is done by taking the first thread id of the first non-empty scheduling deque, searching the context cell from the `threads` trie with the thread

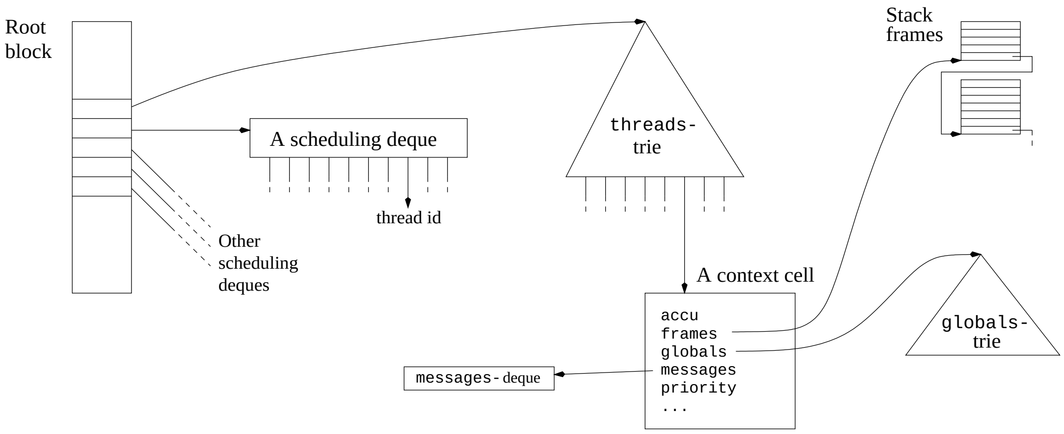


Figure 4.5: Data structures in the byte code interpreter.

id, and reading the values in the context cell into the C variables implementing the virtual machine registers. Yielding a thread implies allocating a new context cell, storing the virtual machine registers in it, inserting the context into the `threads` trie, and inserting the thread id last into a scheduling deque.

The value of the virtual machine register `priority` maintains the number of the scheduling deque while the thread is executing. It is possible that the thread changes the value of `priority` in order to change its priority level.

Spawning a new thread involves finding an unused thread id, allocating and initializing a new stack frame and context cell, storing the new context cell in the `threads` trie and inserting the thread id in one of the scheduler deques. The spawned thread inherits the `globals` trie from the spawner. Killing a thread is accomplished by merely removing it from the `threads` trie. If the thread id is encountered in the scheduler deque at some later moment, it is simply ignored.

A thread is blocked if its id is not in any of the scheduling deques. A `blocked` bit in the context cell is maintained to tell whether the thread is blocked. A thread may block only at the beginning of a byte code block. The `blocked` bit is set in the context cell and the thread id is not inserted in a scheduler deque, but otherwise blocking a thread is identical to yielding. Note that since only the running thread may block, there is no need to remove thread ids from the scheduling deques. Waking up a blocked thread is done by inserting its thread id into one of the scheduler deques and clearing the `blocked` bit in the context cell. The latter operation may, of course, require replacing the context cell in the `threads` trie.

Each thread has a virtual machine register `messages` referring to a deque of incoming messages. Since Shines is a strict functional programming language, the cells in a message can be passed by reference rather than copied. In addition to inserting the message in the deque of the receiving thread and inserting the new context cell in the `threads` trie, we also have to check the `blocked` bit of the receiving thread's context. If the bit was set, we clear it and insert the receiving thread's id into a scheduler deque. The purpose of the `blocked` bit is to prevent the thread from being inserted multiple times in the scheduling deques which would result in an uneven sharing of CPU time among threads of equal priority.

Numerous optimizations are naturally possible. In the next section we show that trie operations and copying context cells is rarely necessary. Furthermore, unless a thread blocks, it is allowed to execute several, typically approximately a hundred byte code blocks before it yields. This reduces the frequency and therefore the average overhead of context switching. Similarly, numerous context switches are allowed to pass before testing for network activity since this involves costly operating system calls. On the other hand, if there are no runnable threads, the byte code interpreter suspends to wait for events from the network, timers, etc.

The byte code interpreter contains numerous features we shall not study further here. Most notably Marko Lyly has been working on profiling and debugging support, both of which add a considerable amount of complexity to the byte code interpreter. Also the Just-In-Time compiler, discussed briefly in Section 4.3.2.7, and possibly in future the support for multiprocessor computers or multithreaded architectures will require new features in the byte code interpreter.

4.2.4 Optimizations

4.2.4.1 Selective Mutations

In Sections 3.4 and 4.2.1 we mentioned that there are optimizations which do mutate cells in-place. In Shades this is possible if the cell resides in the first generation. But we must also retain the correctness criteria of Shines. For example, assume we have a primitive that assigns `accu` the thread's message queue. If that message queue is then updated-in-place by the message delivery mechanism, the Shines program will observe arbitrary changes in the value it was given. This, in turn, breaks referential transparency and in more severe cases may even lead to run-time errors and database crashes. Therefore, whenever a cell "escapes" from the interpreter's exclusive view to become visible in the Shines program, the cell must also become immutable even if it resides in the first generation.

Stack frame cells escape from the interpreter to Shines in two cases. When we refer to a function, the function's prototype stack frame is assigned to `accu`. These stack frames are, however, not yet in execution and since they will be copied during the function call, referential transparency is not jeopardized.

The second case is when the Shines program executes the byte code instruction `call_with_current_cont`. As mentioned earlier, this instruction issues a call to the function f and passes as argument the current value of `frames`. This makes it possible that the call site of f is returned to several times. Furthermore all call sites in the call chain before the call site of f can be returned to several times.

Therefore, whenever a `call_with_current_cont` is being issued, the current stack frame and the stack frames reachable from it by following the `next`-pointers frame must become immutable.

One solution would be to reserve a bit in each stack frame to tell whether it has escaped and mark all stack frames from `frames` to mature generations escaped in the `call_with_current_cont` instruction. Unfortunately this marking would not be particularly real-time. We could, however, devise incremental schemes where we mark a stack frame escaped and copy it if we return to it from a marked stack frame or by using the `return_to_cont` instruction. Furthermore, the Shines compiler might perform escape analysis [12] trying to decide whether a stack frame remains in the program's data set after it has been returned to. Escape analysis could also be used to avoid the copying of stack frames representing closures which will be called only once. Whether or not these techniques are worthwhile is left for future research.

Furthermore, there are instructions which make *all* stack frames, context cells and message queue cells escape. These instructions are used for example in the debugger to get a snapshot of the entire state of the database. Although the stack frames in these snapshots may appear under a different Shines type and may therefore be uncallable, we still would not want them to change arbitrarily when we inspect them in the debugger.

At this stage of research we opted for a simple solution: whenever we issue an instruction which allows cells to escape, we assign the variable `escape_point` the current value of the first generation allocation pointer. Since the first generation is contiguous and allocated downwards, we can test, using a pointer comparison, whether a cell has been allocated after the previous escape. If the cell is newer, it can be updated-in-place, otherwise we have to copy it.

Below we list all the update-in-place–optimizations currently used in the byte code interpreter. The cells are implicitely required to be newer than `escape_point` and reside in the first generation.

- Stack frames are not copied, but updated-in-place when they are returned to after a call. This improved performance by 50% on average.

- All deque cells in message queues and scheduling queues, all context cells, and all trie nodes in `threads` trie are updated-in-place. This improved performance by another 50% on the very communication-intensive benchmark "Client" described in Section 5.1, but little improvement was observed elsewhere.

- If we are about to issue a tail call and the callee's stack frame is at most as large as the caller's stack frame, then we reuse the caller's stack frame as the callee's stack frame instead of allocating a new frame. This improved performance by merely a few percentages.

4.2.4.2 Caches

Although tries are amazingly efficient data structures, an array lookup is considerably faster. We have therefore chosen to augment the `threads` trie and the `globals` tries with software caches. The caches are two-way associative, their sizes are fixed powers of two, and they are indexed by a quickly generated hash value so that lookups can be performed in only approximately ten machine instructions. For example the hash key for the threads cache is simply the thread id modulo the cache size. In the globals cache we also have to regard the thread id as well as the interned string id of the variable as the key.

These caches are neither part of the root set nor are they allocated in the database image managed by Shades. Therefore they will have to be cleared every time Shades collects garbage. Since we furthermore wish these software caches to mostly remain in the processor cache for efficiency, we chose these software caches to be rather small, usually only a few hundred lines. Nevertheless they have a surprisingly high hit ratio and provide measurable, although not dominating, speedups of approximately 5% on average.

Software caching has been found useful also in some other places, for example in caching the results of dynamic subtype tests between two objects.

4.2.4.3 Byte Code Dispatching

Compared to index structure operations, disk or network IO, or message passing, dispatching from one byte code instruction to the next byte instruction is relatively cheap. But for very small instructions dispatching costs dominate. Just-In-Time compiling, as described in Section 4.3.2.7, essentially removes all dispatching, but it has its own drawbacks. Therefore, some effort has been taken to make the byte code dispatcher reasonably efficient.

The simplest way to write a byte code dispatcher in C is to use `switch`-statements. Each instruction is implemented as its own `case` as follows:

```
while (1)
  switch (*pc) {
  case INSN_load_imm:
    accu = pc[1];
    pc += 2;
    break;
  case INSN_add:
    accu += *--sp;
    pc++;
    break;
  ...
  }
```

Here we assume instructions are represented as successive integer values of an enumerated type.

While this is portable ANSI C, it is not the most efficient instruction dispatcher since the `switch`-statement must perform a redundant range check for `default` cases or values not present in the cases. Furthermore, the `break`-statement jumps back to the beginning of the loop.

We use the GNU C extension known as "labels as values" whenever possible. With this extension it is possible to take the address of a C label and store it in an array. Since C does not perform array bounds checking, we can eliminate the redundant range check. Furthermore we move the centralized dispatching directly to the end of each instruction, thereby avoiding the jump back to the beginning of the loop. Our dispatcher is implemented as follows:

```
void *jump_table[] = {
  &&LABEL_load_imm, &&LABEL_add, ...
};
...
goto jump_table[*pc];

LABEL_load_imm:
  accu = pc[1];
  pc += 2;
  goto jump_table[*pc];
LABEL_add:
  accu += *--sp;
  pc++;
  goto jump_table[*pc];
```

Entering the instruction dispatcher is now done with the first `goto` on the fourth line.

This dispatching still includes the overhead of reading the `jump_table` array, which should nevertheless be rather small as it should usually fit in the fastest cache memory. In future we intend to store the labels directly into the byte code cell so that dispatching can be done as efficiently as `goto *pc`. This, however, is not yet supported by Shades as it requires that byte code cells must be initialized with the correct label values during recovery.

Even more efficient dispatching is possible, e.g. the *memcpy* idea presented in [76] and later in [72]. Both articles in addition to [22] also serve as wider surveys of instruction dispatching.

4.3 The Byte Code Compiler

4.3.1 Bootstrapping

The history of bootstrapping Shines is rather unique and probably deserves a mention. Rather peculiarly the only program that produces byte code assembler is written in Shines itself!

The standard technique to bootstrap a compiler for a language X is to first write a bootstrap compiler for X in some other language. We call the executable binary of this bootstrap compiler X_{boot}. Then the compiler for X is rewritten in X itself and compiled with X_{boot} to obtain an executable binary X_1. Usually the bootstrap compiler is as simple as possible and contains hardly any optimizations. Therefore the compiler is recompiled, now with X_1 instead of X_{boot}, to obtain a hopefully more efficient executable X_2. For testing purposes we may compile the compiler with X_2 and obtain X_3, which should now be equal to X_2.

The development of the Shines compiler began before either Shines or the byte code interpreter had sufficient string primitives or means to parse incoming data. Yet we had an urgent need to produce efficient byte code programs. Had we merely implemented a simple bootstrap compiler we could have wound up with very pessimistic benchmarks and misleading profiling results. On the other hand, every moment spent on implementing optimizations to the bootstrap compiler is wasted effort as the bootstrap compiler will have to be eventually discarded in any case. Furthermore implementing the Shines compiler in Shines gives important feedback to the design of Shines itself.

We began bootstrapping Shines by writing, in Objective Caml, a simple tool which translates Shines code to Scheme. We use the C preprocessor to give a vague imitation of a module system. The same tool can be used to read a Shines file and produce Shines code, which when executed builds a syntax tree as if it were lexed and parsed from the original file given to the tool.

Compiling a file into byte code assembler can now be done as follows: First we translate the file into Shines code which builds the syntax tree of definitions in the file as if they had been parsed by a compiler front-end. Then we catenate that code with the compiler backend written in Shines and translate the result into Scheme. Finally we execute the Scheme code, and if all goes well, the Scheme interpreter will eventually write the byte code assembler into its standard output stream.

This worked and served as a temporary solution for several months. But the generated Scheme code was both abundant and suboptimal, and eventually, as the Shines compiler grew more complex, hundreds of megabytes of RAM and hours of CPU-time was required by the Scheme interpreter to compile the compiler. At this point Shines did have sufficient string and network primitives, but still lacked tools for lexer and parser generation. The next step was to convert the old bootstrap tool to translate the Shines source into an easily parseable format. A simple-minded recursive descent parser manually written in Shines reads the output and constructs a syntax tree fed to the backend as before.

Currently the Shines compiler compiles approximately 7000 characters of preprocessed source code per second on a PentiumII at 350 MHz. It compiles itself and various required library code, some 10.000 lines in total, in less than a minute.

4.3.2 Passes and Optimizations

Lexing and parsing are currently done by the boot compiler, and type checking is not yet done at all.

4.3.2.1 Desugaring

The first few passes of the compiler expand syntactic sugar. For type definitions this includes various preprocessing and checking inheritance hierarchies and generating code to instantiate prototypes.

Patterns are a convenient way to write complicated tests over the structure of a value and simultaneously bind parts of the value into local variables. For example a function accepting a list of tuples and returning another list containing the first values of the tuples could be written as

```
unzip_left(list) :=
  MATCH list
  ON [<left, right>] ++ tail GIVE [left] ++ unzip_left(tail)
  OTHERWISE []
  END;
```

In Shines, brackets denote lists, double plus-signs list appending, and angles (`<` and `>`) denote both inequality comparisons in addition to tuples.

For pattern matching desugaring means expanding patterns into various nested tests, conditions and local variable definitions. For example the above pattern expands to as many seven local bindings in addition to actual tests. Fortunately the let-hoisting optimization presented in the next Section will substitute all but the one holding the tuple in the head of the list.

Currently each pattern in a match expression is compiled separately, although some efforts have been taken to generate more efficient code from them. The standard solution would be to construct a decision tree [7] or an automaton [69] from all the patterns. Vera Izrailit came up with an alternative idea of sorting similar patterns adjacently so that some simple algebraic optimizations similar to common subexpression elimination in later passes can remove redundant tests. This will be discussed in further detail in her Master's Thesis.

A *stream* is an ordered sequence of values evaluated on-demand. It is implemented as syntactic and type sugar for a nullary lambda function which returns a tuple consisting of the first value in the stream and the rest of stream. The end of a stream is an empty tuple. For example the Shines function

```
int_to_stream(first, increment) :=
  \() <first, int_to_stream(first + increment, increment)>;
```

returns an endless stream containing all natural numbers starting from `first` with `increment` increments. The expression `\(a) b` creates an anonymous lambda function taking the argument `a` and returning the value of expression `b`.

Stream expressions are desugared to a complicated structure of lambda expressions. For example the expression

```
{ <v,u> | v IN sa, u IN sb, v < u }
```

contains all tuples of the cartesian product of streams `sa` and `sb` satisfying the inequality. It would be desugared into the equivalent of

```
LET g1(s1,g2) :=
      MATCH s1()
      ON <> GIVE <>
      ON <v,s2> GIVE
        LET g3(s3,g4) :=
              MATCH s3()
              ON <> GIVE g2(s2,g2)
              ON <u,s4> GIVE
                IF v < u
                THEN <<v,u>, \() g4(s4,g4)>
                ELSE g4(s4,g4)
              END
        IN g3(sb,g3)
      END
IN \() g1(sa,g1)
```

Note that the Y-combinator has been used twice to create mutually recursive local functions. In Shines the `LET`-binding `f(x) := ...` is syntactic sugar for `f := \(x) ...`

4.3.2.2 Algebraic Optimizations

Algebraic optimization is a general term for numerous traditional optimizations performed with local rewrites in a bottom-up manner on the syntax tree. These include constant folding of integers and booleans, simple common subexpression elimination, some strength reductions, removals of redundant computations, various related canonicalizations, and in our case also let-hoisting.

Let-hoisting (see [7, pp. 93-98] and [43]) tries to minimize the scope of the local variables, and if the value of the variable is very easily computable or the variable is used only once, the value may even be substituted for the variable. This optimization roughly halves the size of the stack frames and almost equally dramatically reduces the number of instructions.

Algebraic optimizations are usually expressed elegantly with simple rewrite rules using patterns. For example the rule

```
ON Plus.<left = Int.<l := value>, right = Int.<r := value>> GIVE
  Int.<l + r>
```

is one of the six rules which fold integer constants in addition. Before applying the rewrite rules the subtrees are analyzed for various features: the uses-analysis tags each syntax tree node with the set of local variables the expression refers to, effect analysis maintains (pessimistic) information on what kind of side effects and interdependencies two expressions can have, and crude uniqueness analysis tells (again pessimistically) whether the reference to the value of the subexpression must be the only reference to the value, thereby possibly allowing updates-in-place to the value as described in Section 3.4.

Immediately after the algebraic optimization pass we lift lambda functions into global functions and prepend their argument lists with the variables belonging to their closure.

The compiler does algebraic optimizations twice, once before and once after function inlining. The post-inlining optimizations are done because inlining often reveals new opportunities for optimizations. The pre-inlining optimizations are done because the choice of whether to inline depends on the sizes of the inlined and calling functions and algebraic optimizations can dramatically change the size of the syntax trees. Deciding whether or not to inline follow the ideas presented in [81]. The inliner was written mostly by Antti-Pekka Liedes.

The virtual machine uses a stack, but because of `accu`, it is not a pure stack machine. Many instructions, e.g. branch instructions, which pop values away from the stack leave `accu` unaffected. This can be used to optimize code. Consider the expression `IF x > y THEN y + 1 ELSE y`. The branch instruction, regardless of whether the branch was taken, leaves `y` in `accu` and it is therefore useless to recompute it in the `THEN`- and `ELSE`-branches. In this pass the compiler traverses the syntax tree keeping in mind the subexpression(s) computing `accu` and replacing subexpressions with no-ops should they recompute `accu`. Proper attention is naturally placed on side effects and branches.

4.3.2.3 Linearization

Linearization is the rewriting of the syntax tree into a list of instructions.

The first pass of linearization decides the positions of local variables in the stack frames. The variables' values can later be read by picking the corresponding value from the stack frame.

The next pass, the uses-after–analysis, finds out which local variables in the scope are no longer referred to in the execution path of the program. This information is used in the next pass to annotate byte code blocks with descriptions of whether the values in the stack and `accu` are used in the rest of the function. This allows Shades to collect away values of local variables which are still in the scope but are useless for computation. This optimization, also called *live-precise* garbage collection, has been reported to reduce heap size by an average 11% for a suite of Java benchmarks, with most programs showing some difference and a few showing more dramatic differences [2]. We chose to implement this optimization because main memory databases usually manipulate large data sets and performing the above optimization manually is very difficult in a functional language.

The actual linearization is done after the above passes. It has to generate dedicated instructions for tail calls. In addition to generating actual byte code instructions, it generates various description codes which describe the state of the run-time stack and `accu` before the instruction. They are used as a basis for debugging, profiling, Just-In-Time compiling, and live-precise garbage collection described in the previous paragraph. An additional pass over the byte code instruction block computes the maximum depth of the stack required to run the byte code block.

4.3.2.4 CPS Conversion

Even disregarding description codes, linearization does not always generate real byte code instructions, but it occasionally uses "pseudo" instructions. These pseudo instructions assume it is possible to block, use unlimited amounts of time, or issue function calls so that they return to the next instruction after the pseudo instruction. Using pseudo instructions in linearization makes it easier to generate code for some complex instructions, function calls, etc. In reality, however, the only place to enter (or return to) a byte code block is the beginning of the block.

Conversion to Continuation Passing Style (CPS) splits the given byte code block at pseudo instructions into multiple smaller byte code blocks, e.g. the block before the call and the block returned to after the call. In the process it replaces pseudo instructions with their corresponding "hard" instructions and surrounds all converted byte code blocks with appropriate descriptions, headers and other information.

An example of CPS conversion is given in Figure 4.6. Our algorithm traverses the unconverted block from its last instruction to the first. When it encounters a pseudo instruction it converts it to a hard instruction which returns to a byte code block containing the instructions reachable from the succeeding instruction of the unconverted block. Note that in order to prevent code explosion we reuse already converted code blocks lowermost in the figure.

Since byte code blocks are the smallest unit of allocation and scheduling, excessively large byte code blocks lead to unfair scheduling and high minimum first generation sizes. Therefore we insert pseudo no-op instructions in suitable places of

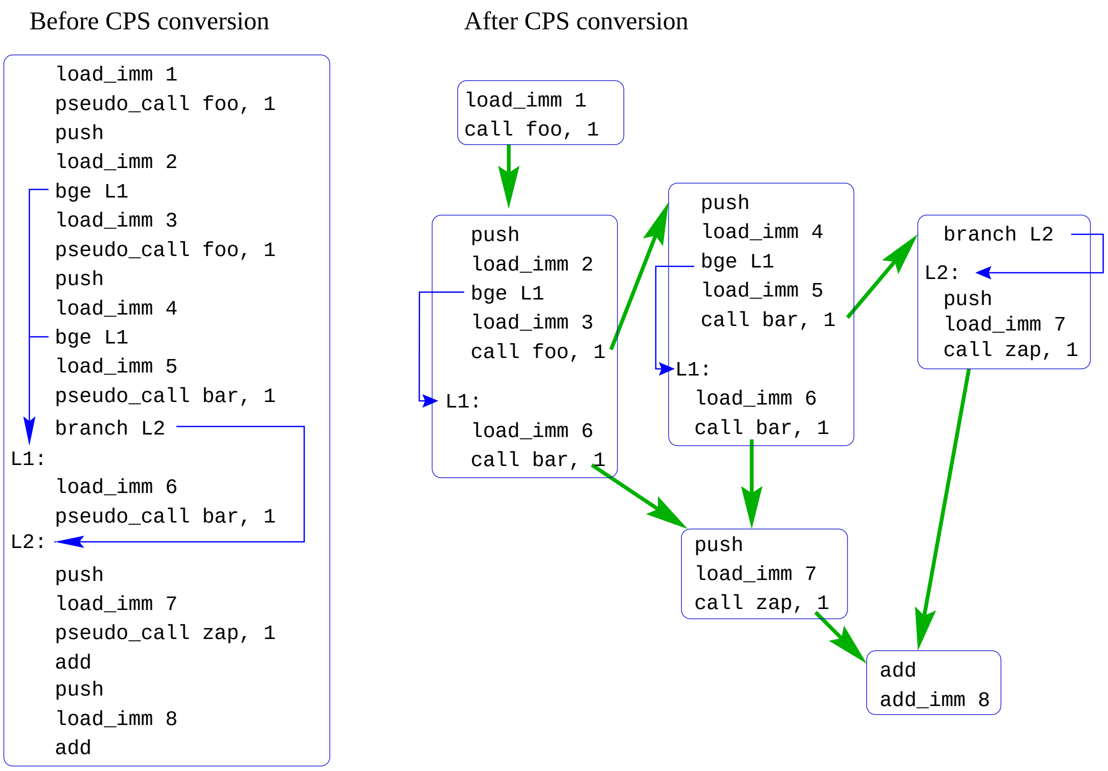


Figure 4.6: The byte code blocks of the expression `(IF foo(1) < 2 AND foo(3) < 4 THEN bar(5) ELSE bar(6)) + zap(7) + 8` before and after CPS conversion. Thin lines represent jumps within a byte code block, thick lines represent byte code blocks returned to after the calls. Peephole optimizations will remove the redundant branch to `L2` in the rightmost byte code block.

large byte code blocks and rerun them through CPS conversion. Typically we split blocks of 150 instructions further into smaller blocks.

Our algorithm for CPS-conversion differs from Appel's [7, pp. 55–66] also in some other respects. For example, we convert linearized byte code blocks while Appel converts lambda expressions to CPS.

4.3.2.5 Peephole Optimizations

The peephole optimizer examines a handful of instructions at a time and tries to replace those instructions with a faster sequence. Typically the instructions are either consecutive or one is the branch target of another. Peephole optimizations include removal of redundant instructions (e.g. a push instruction before function return), flow control optimizations (e.g. branches to unconditional branches), and most visibly heavy use of machine idioms (e.g. conditional loads). Currently our peephole optimizer has approximately a hundred rewrite rules, again written nicely using pattern matching.

Since one peephole optimization may reveal opportunities for new peephole optimizations, each peephole optimization pass is performed until no more improvement was achieved, though at most ten times. Each peephole optimization pass is surrounded by a pass of dead code removal.

Peephole optimizations are done both before and after CPS conversion, because the CPS conversion can both hide and reveal opportunities for peephole optimizations.

Finally, immediately before printing out the final byte code blocks, one additional peephole pass is performed. This pass combines some frequently occurring phrases of very simple instructions into single instructions, thereby reducing the instruction dispatching cost in the byte code interpreter. As mentioned in Section 4.2.1, one frequent idiom is the sequence of instructions `pick`, `push` and `pick`.

4.3.2.6 Byte Code Assembler

The byte code assembler reads the textual representation of the byte code and creates corresponding byte code blocks in Shades. Even the byte code assembler performs some optimizations. For example, it tries to convert symbolic references to objects into direct pointers to the objects. Instead of looking up interned string identifiers in the instruction `load_imm_string`, strings are accessed through direct pointer to the string. Instead of looking up the stack frame prototype through the global variable trie, the stack frame prototypes are usually accessed through pointers in various call instructions. This pass can even build cycles of pointers: byte code blocks and their stack frame prototypes of mutually recursive functions referring to each other as long as these cycles do not span over generation boundaries.

The byte code assembler does also some very low-level peephole optimizations. For example calls to global functions of arities zero to four are specialized to dedicated instructions. Push instructions before another instruction are melded into the succeeding instruction. This is conveniently implemented in the byte code interpreter using C as follows:

```
case INSN_push_and_load_imm:
  *sp++;
case INSN_load_imm:
  accu = pc[1];
  pc += 2;
  break;
```

Note that there is no `break` statement in the first case. This simple technique avoids the dispatching cost of almost all push instructions and reduces average CPU consumption by over 10%.

Naturally the byte code assembler also takes care of finding branch targets, precomputes the maximal memory consumption of the byte code block, etc.

4.3.2.7 Just-In-Time Compilation

Interpreting optimized byte code with an efficient dispatcher can be surprisingly performant. Frequently we have witnessed cases where most time was consumed in index structures or network or disk IO. In some applications, however, most time was consumed in the byte code dispatcher or the byte code instructions themselves. The cost of dispatching can be eliminated by compiling byte code programs to native machine code. Also some further optimizations become possible: e.g. immediate values of byte code instructions can be propagated to immediate values in the native code. They were previously read pc-relatively from the byte code block as in the `load_imm` instruction in the previous section. Similarly all stack pointer processing at run-time can be omitted.

Writing a compiler to native code is, however, a major project, particularly if several processor architectures have to be supported. Therefore we chose to use the C compiler: we emit the C code implementing the byte code instructions for each instruction in the block, compile the resulting C file to object code and link it with the database server. The compiled byte code blocks appear as C functions and they are called by some trampoline code in the interpreter.

Preliminary tests indicated that highest speedups, over 400%, were obtained for very small number crunching programs such as computing Mandelbrot fractals. For many programs the speedups were more modest, negligible or even negative. Negative results can largely be attributed to instruction cache thrashing since byte code tends to be more concise than native code compiled from C sources [19]. Furthermore compiling the C code for some of the largest byte code programs took a prohibitively long time.

Our current effort is to be selective about the byte code block we compile. We plan to compile only the most frequently used byte blocks containing mostly “small” instructions which perform arithmetic, branching, function calling, list and tuple processing. The resulting C-code files are compiled concurrently and linked dynamically with the server. This makes the byte code interpreter a Just-In-Time (JIT). The idea of JIT compiling has been known for decades under various names, but has recently been popularized by Java systems.

We also try to emit more idiomatic C code, as if it were written by a human programmer, since C compilers seem to be tuned to produce better code for such typical programs. Some annotations generated in the linearization process are used to retain local variables of Shines also as local variables in the C code; if they were pushed and popped from the stack they could hardly be register allocated by the C compiler.

The JIT compiler is being written by Jukka-Pekka Iivonen with occasional consulting by yours truly. Although some benchmarking is currently possible, numerous important details and interesting ideas remain to be implemented.

Regardless of JIT, it is hardly possible to achieve performance higher than well crafted C code — after all Shades, indexes and the byte code interpreter were also written in C. But in many applications Shines can come close to optimal C. In some areas, particularly in concurrent real-time programming, in reliable computing, and in manipulating large persistent data sets, Shines will probably be much more efficient than what even a competent C programmer would implement. And no C programmer has the luxury of built-in persistence, garbage collection, first class functions and continuations, expressive type systems, etc.

Chapter 5

Experiments

In this Chapter we present some experiments and benchmarks on our database server. The role of the experiments is *not* so much to benchmark our database server against existing solutions and commercial products. Comparison would be impossible in any case because practically no established benchmarks for main memory databases exist. Rather, we wish to understand better the behavior of our server, find recommendable values for various control parameters, and discover performance bottlenecks and areas of further work. Most results should be regarded as tentative as our database server is still being actively developed.

5.1 Benchmark Programs

To date only one comparative benchmark has been conducted on Shades and its index structures: early in 1998 Nokia Telecommunications (NTC) defined an Intelligent Network database workload and invited representatives for Oracle 8.0.4, Objectstore 5.0, and HIBASE to conduct the benchmark on a large mainframe at NTC. The database contains four static tables and one updatable table, totalling to several gigabytes in size. The transactions occasionally updated a counter associated with the given telephone number, but mostly translated an incoming telephone number to another telephone number depending on the day of the week and time of day. Shades outperformed its commercial rivals by approximately a factor of ten although Shades could utilize only one of the 12 available processors.

We refrain from discussing the Intelligent Network benchmark further in this Thesis because numerous details are probably confidential, the benchmarks are no longer repeatable due to absense of the rivals and machinery, and because the benchmark utilizes data structures dedicated to telephony. A separate, but narrowly circulated, report describes the benchmarks and results in detail [54].

TPC-B is a commonly used and carefully specified benchmark for databases [30]. It simulates money transfers from one account to another in a banking environment. Each transaction requires three updates in three tables of varying sizes and an append to a history table.

But unfortunately a conformant TPC-B benchmark requires very large tables, making it impossible to use in main memory databases. We have therefore discarded the history table, various pad columns in the tables, and we regard the number of rows in tables as a freely adjustable parameter. Each "tps" in this database size parameter requires approximately 1.2 MB of logical data, but additionally almost a megabyte is spent on cell headers and index structures. Up to 30 tps fits in our database image, but since almost all time would then be consumed in garbage collection, we use the tps-values 10 and 20 in the benchmarks.

The result is given in transactions per second. The CPU-time required to generate the input arguments for the transactions is excluded from the transaction processing time.

System M was informally described by Garcia-Molina and Salem in conjunction with their transaction processing testbed [77]. After being generously allowed to peek at their source code we were able to reproduce the benchmark. The benchmark simulates a credit card application with four tables of varying sizes and eight transaction types, most of which perform one update.

We noticed that we could fit more data in Shades than System M even if the machine we used has as much memory as Garcia-Molina's and Salem's machine. We therefore tripled the amount of logical data to get a second version of the benchmark.

The following benchmarks were written using Shines. Their purpose has never been to compare H$^{\mathrm{I}}B_{\mathrm{A}}$SE to some other systems, but to learn more from the behaviour of H$^{\mathrm{I}}B_{\mathrm{A}}$SE. No Just-In-Time compilation was used.

Compiler is the Shines compiler compiling itself six times. This benchmark involves matching and creating objects, list processing, and a little string processing in addition to typical arithmetic and logic operations and function calls.

AVL performs a mix of various AVL-tree operations on integer-keyed trees. Approximately two thirds of CPU-time is spent in simple find, insert and delete operations, the rest in joins, cuts and cursor operations. Two different data sets, 3000 and 300 000 key-value pairs, were used.

Mergesort is a higher-order list sorting function applied to lists of integers. The benchmark program also generates the lists and checks for correctness of the sorting. Three list lengths, 100, 10 000, and 1000 000 were used in the benchmarks. The benchmark was repeated a desired number of times.

Client contains a server thread discussing with either 100 or 1000 client threads. The messages are simple tuples and little processing is involved in generating and interpreting them. This benchmark, however, incurs a heavy load on context switching and messaging primitives described in Section 4.2.3.

The three first benchmarks (Intelligent Networks, TPC-B, and System M) were implemented in C mostly by Kengatharan Sivalingam. The AVL benchmark was written by Jarkko Lavinen. Several authors have participated in writing the other benchmarks.

In the following sections we shall vary several parameters of Shades, including first and mature generation sizes. However, 70 MB was always reserved for mature generations in the following two sections, and 300 MB in Section 5.4. Furthermore, denoting first generation size with f, we fix *mid_gc_limit* to $2f+1$ MB, *start_gc_limit* to $2f+3$ MB, and *mid_gc_effort* to 5 MB. These three parameters have little effect on throughput, but as described in Section 2.7.3, they do affect commit group latencies. These values represent a conservative choice.

5.2 Object Lifetimes

The efficiency of generational garbage collectors is based on the strongly evidenced fact, the *weak generational hypothesis* [88, 42], that an overwhelming majority of objects die young. In his garbage collection survey, Wilson states that typically 80 to 98% of objects die before one further megabyte of heap storage has been allocated [91]. In a highly optimized Haskell compiler, no more than 5% of objects survive beyond one megabyte of allocation [78]. Measurements of ML-programs in SML/NJ conclude that only 2% of objects in youngest generation typically survive the first collection [5].

We have little doubt that we will end up with very similar statistics if we use Shades only as a memory manager of a functional programming language. However, Shades will be also used for databases, which conceivably have a somewhat different behavior. In order to test the weak generational hypothesis in databases we counted the average number of bytes surviving the first generation collection with varying first generation sizes in all our benchmark programs. The results are summarized in Figure 5.1.

Although most objects die young also in database applications, a considerably higher ratio of objects do survive the first generation collection than in previously reported statistics: For a large fraction of applications and typical first generation sizes we observe that 15%–30% of objects survive the first generation collection and are therefore also written to disk.

Except for the smallest first generations, the TPC-B benchmarks have the highest survival ratio. These benchmark programs were written in C and essentially all the cells they allocate are either trie nodes or tuples containing the records of the

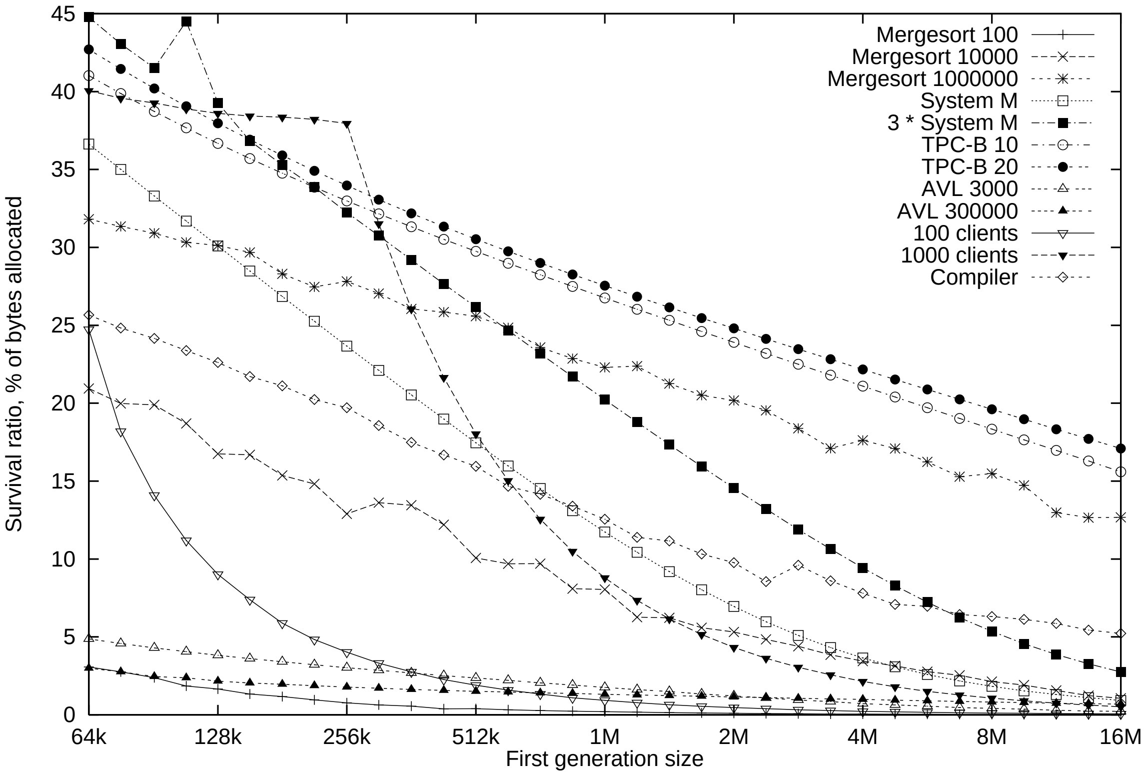


Figure 5.1: First generation survival ratios.

tables. The levels close to the root of the tries are copied several times in one first generation while leaves are copied several times only rarely.

The benchmarks on sorting lists show how strongly the size and longevity of the data sets affects the survival ratio. A list of one million elements can hardly be allocated in a large first generation whereas a list of 100 elements can be created, sorted, and tested several times in all but the smallest first generations. It can also be noted that the versions TPC-B, System M and Client benchmarks with larger data sets have a higher survival ratio.

The "1000 clients" benchmark shows how dramatically mutations in the message and thread system affect survival ratios. For very small first generations most cells must be copied since a client thread gets executed only once if at all in each first generation. But when the first generation grows large enough to allow for a second time slice for the thread, its context cell, `threads` trie nodes, message deque cells, and stack frames need not be copied. This in turn slows down first generation consumption so that cells are much older and much less likely to be alive when the first generation is eventually collected. Furthermore, since long-lived context cells

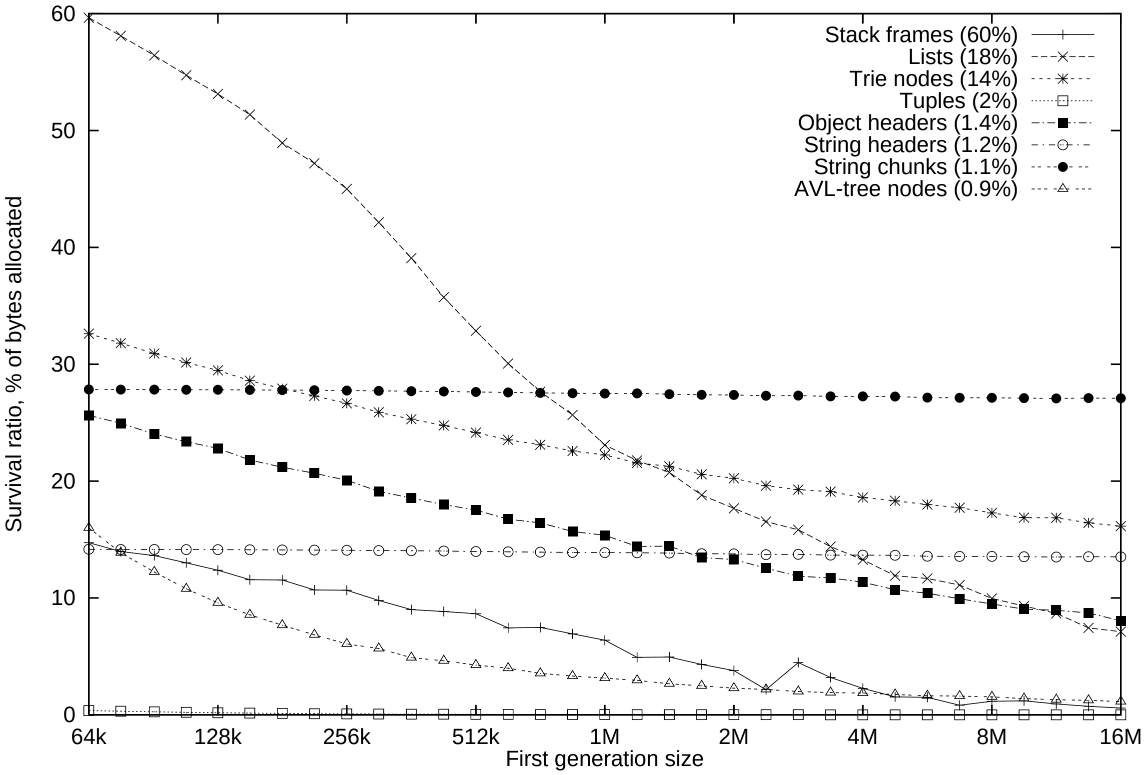


Figure 5.2: First generation survival ratios of each cell type in the compiler benchmark.

no longer need to be copied, a larger fraction of allocated cells are more short-lived. Hence the rapid drop in survival ratio.

Surprisingly the AVL tree benchmark has a very low survival ratio although at the first glance it should behave like the other benchmarks, TPC-B and System M, which also manipulate trees. A closer look reveals that the AVL-tree nodes do have a relatively high survival ratio, ranging from 14.5% to 4.7%, but only 14% of the bytes allocated are in AVL-cells. Most, 78%, of the cells are stack frames with 0.9% to 0.003% survival ratio as they are alive only for the duration of a key comparison or a single AVL tree operation. We assume compiler optimizations based on escape analysis and correspoding byte code support might reuse most of these stack frames.

Next we shall take a closer look at the compiler benchmark. In particular we wish to see which cells survive the first generation, which die young. These results are given in Figure 5.2. The percentages after each cell type denote their fraction of allocation.

While the stack frames in the AVL benchmark were very short-lived, the stack frames in the compiler have a survival rate from 15% to 0.5%. This is explained by the length of call chains in the peephole optimization and CPS conversion passes. For example the peephole optimizer makes a non-tail-recursive call for each byte code instruction it produces. Before CPS conversion these recursions can be as deep as tens of thousands of calls. A first generation collection during such a call promotes the whole stack frame chain to a mature generation.

Lists behave somewhat similarly to stack frames. All compilation after CPS conversion to emitting byte code assembler can be done in one large first generation. Since CPS conversion splits excessively long byte code sequences, the survival ratio of lists drops rapidly in the range from 256 kB to 2 M.

Objects and tuples have a very different survival ratio. The dominant programming practices use tuples very locally for example in returning multiple values from a function call or for pattern matching several values simultaneously. Objects, on the other hand, are used in wider contexts to construct syntax trees, typing and side effect information, etc.

Strings can be either very short-lived, created only as intermediate results for printing or for lexing and parsing the source code into a syntax tree, or they can remain live through most of the compilation in case they should be needed for informative error messages or for emitting string constants in the byte code assembler.

Although not included in the figure, byte code blocks and string interning cells are practically immortal. The metadata cells of Shades are collected away only in mature generation garbage collection, and therefore also have a 100% survival ratio.

Clearly survival ratios tend to decrease as the first generation grows, but the decrease is neither smooth nor monotonic. A higher survival ratio implies more disk writing per transaction. A larger first generation size, on the other hand, implies longer commit group duration and may eventually hamper real-timeness. Whenever the server's throughput is disk-bound and not dominated by mature generation garbage collection, first generation size is the most important parameter to adjust the balance between throughput and real-timeness.

Making some rather daring simplifications we could even calculate the maximum first generation size analytically: given a latency requirement of l seconds, disk bandwidth of b bytes per second, computational and root block writing cost of c seconds, and survival ratio of s, the maximum first generation size is $(l-c)b/s$. For example, assuming a 30% survival ratio, 6 MB/s disk bandwidth, with a 30 ms computational and root block writing cost, and a 100 ms latency requirement, we could use up to a 1.4 MB first generation size. In reality, of course, the first generation size affects survival ratio and computational costs in rather unpredictable patterns. We would also have to consider the disk writing caused by mature generation collection unless it can be done entirely in parallel with computing the next commit group.

We typically use first generation sizes from 512 kB to 3 MB. For the 20 tps TPC-B benchmark this results in 160 kB to 730 kB of disk writing for each commit group.

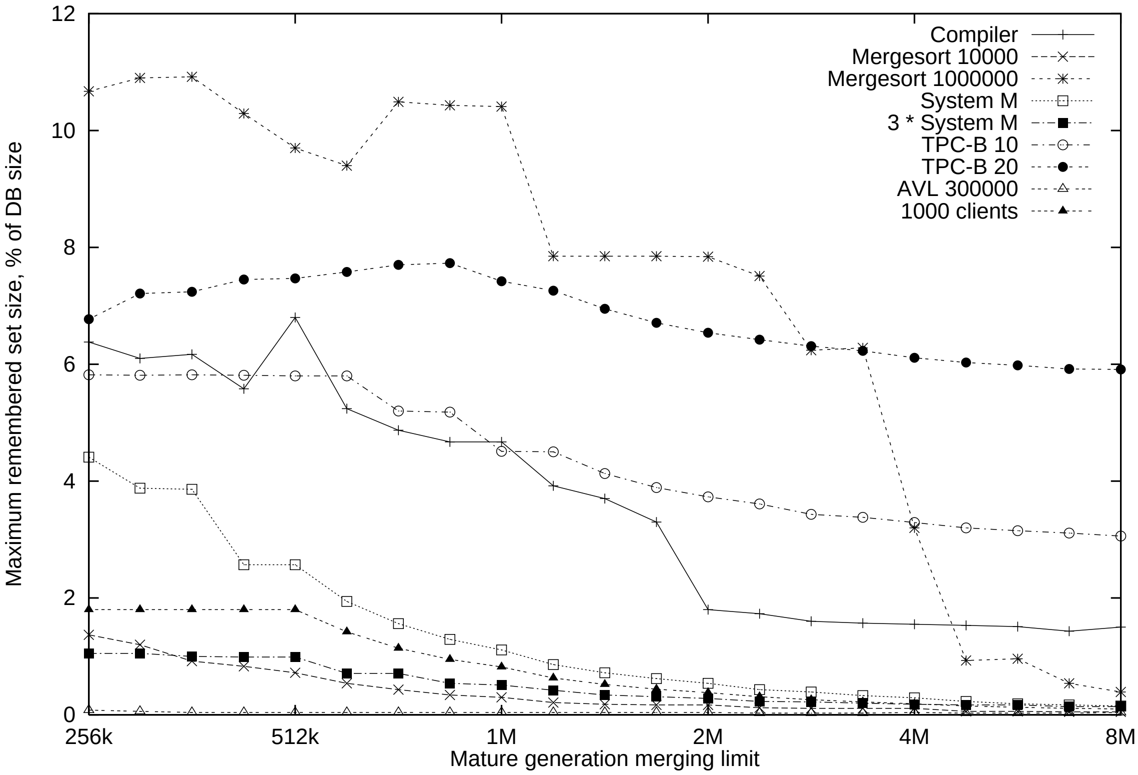


Figure 5.3: Maximum remembered set sizes in various benchmarks.

5.3 Sizes of Rembered Sets

The maximum size of the remembered sets is mostly dependent on the application, i.e. the topology of the graph of live cells in the database. In the worst case almost all words in the database contain a pointer to the same cell c. In such a case the remembered sets will grow to contain almost all the words in the database until the cell c is collected.

Furtunately remembered sets are much smaller in practice. To test this we rerun the above benchmarks and memorized the maximum memory consumption of all remsets combined. The worst observed overhead was 11% for mergesorting. No other test had a remset overhead of over 8%. The tests with negligible remembered set consumption were left out of Figure 5.3.

We used a fixed 1 MB first generation size and varying mature generation merging limits: whenever adjacent mature generations consume together less memory than allowed by this limit, the first line in MAJOR-COLLECTION-STEP in Shades takes them into the same *from_gns*-set and creates only one new mature generation from them.

If we increase mature generation size dramatically we will notice a moderate decrease in remembered set sizes because relatively fewer pointers refer across generation boundaries. Unfortunately overly large mature generations hamper real-timeness similarly to overly large first generation sizes, and this approach can therefore not be recommended. In future we shall fix mature generation merging limit to be 70% of first generation size.

When a first generation is collected and it becomes a new mature generation, all the pointers from that mature generation to uncollected mature generations are inserted to the older generations' remembered sets. Hence, remsets tend to grow also if mature garbage collection advances very slowly compared to normal first generation collections. To prevent this we have set a minimal speed of one collected mature generation per commit group, as mentioned in Section 2.7.3. By having very large mature generations and small first generations we could further this effect, but this can have a negative impact on both real-timeness and performance.

Other parameters have been observed to have little or no effect on the size of the remembered sets.

5.4 Latency and Throughput

We shall now turn our attention from internal behavior of Shades to external behavior, affected by hardware and measured in wall clock time. We shall mostly concentrate on the TPC-B benchmark, vary its data size, first generation size and means of disk IO measuring both throughput and commit group duration.

All benchmarks in this Sections have been executed on a dual 350 MHz Pentium II -based PC with 512 kB internal L2-cache, 512 MB of main memory, and four 4.5 GB Quantum Viking II 4.5WLS 7200 rpm Ultra2-SCSI disks. The benchmark machine cost approximately 30.000 Fmk, almost $5.000, including 22% VAT in the end of 1998. As the operating system we used various versions of Linux.

Except for the root block, all disk writing in Shades is done in pages. Disk throughput increases as page size increases due to the decreasing overhead of seeking, rotational latency, and bus arbitration per written byte. However, based on our experiments, disk performance improves little for page sizes larger than 32 kB. Furthermore, since generations contain an integer number of pages and the last page in the generation is on average only half-full, we prefer not to use excessively large pages. Therefore all the benchmarks in this section were run with 32 kB pages.

We have used a 300 MB database size in the tests of this section. We can fit a database of approximately 130 "tps" in this memory, although for the largest data sets almost all time will be consumed in mature generation garbage collection. All tests were executed for five minutes so that all mature generations in the database had to be collected several times.

As mentioned in Section 2.7.4, disk writing may be a dominant factor in overall performance and it can be speeded up by parallel and asynchronous writing. The benefit of asynchronous writing varies, partly because some disks can supply some asynchrony themselves by employing a write buffer which they can guarantee to write to disk even if a power failure occurs, and partly because several operating systems, including Linux, still have considerable problems with multithreading. Nevertheless, in our benchmark machine pthread-based asynchronous writing produced the speedups shown in Figure 5.4 and 5.5. For comparison we also included a selection of varying number of disks and a transient database, which performs no disk IO, only CPU work.

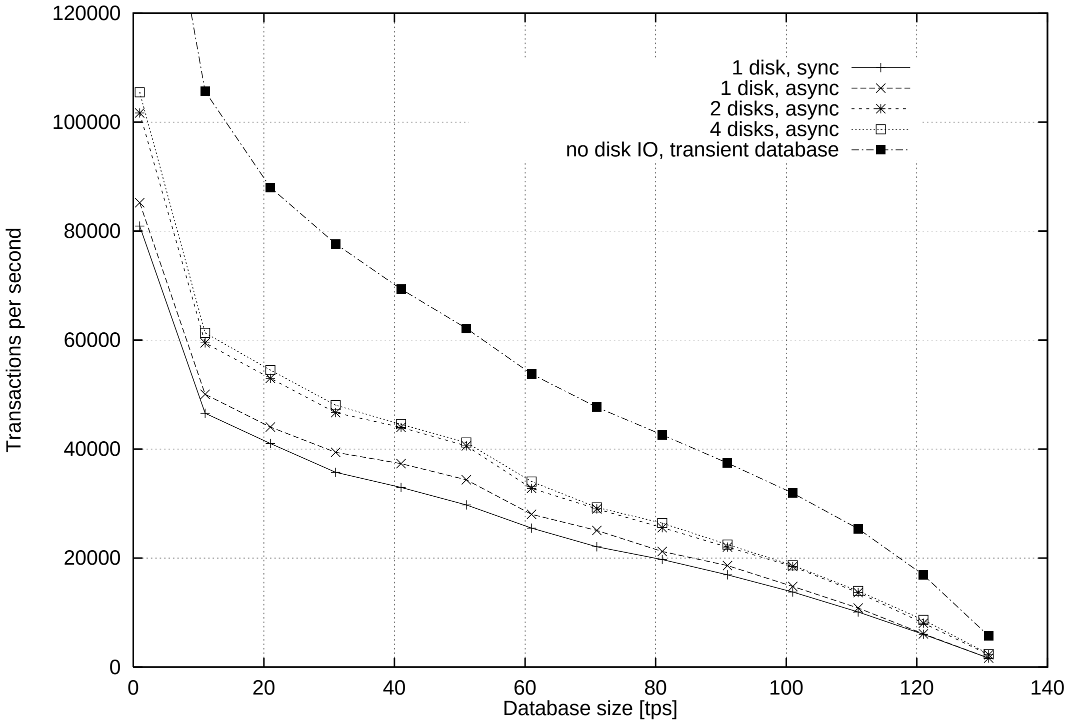


Figure 5.4: The effect of database residency and IO mechanisms on throughput.

Asynchrony is beneficial when either several disks or the disks and the CPU can work concurrently in parallel. When we have several disks we indeed observe a speedup of 25% to 45%. When the database is full, mature generation collection causes more writing, and the relative importance of asynchrony grows. Furthermore, as discussed in Section 5.2, survival ratios are slightly higher for full databases. This increases disk writing per commit group, further emphasizing the importance of asynchrony.

When only one disk is involved, the benefit of asynchrony is small for almost empty or almost full databases. Only when the amount of IO caused by mature

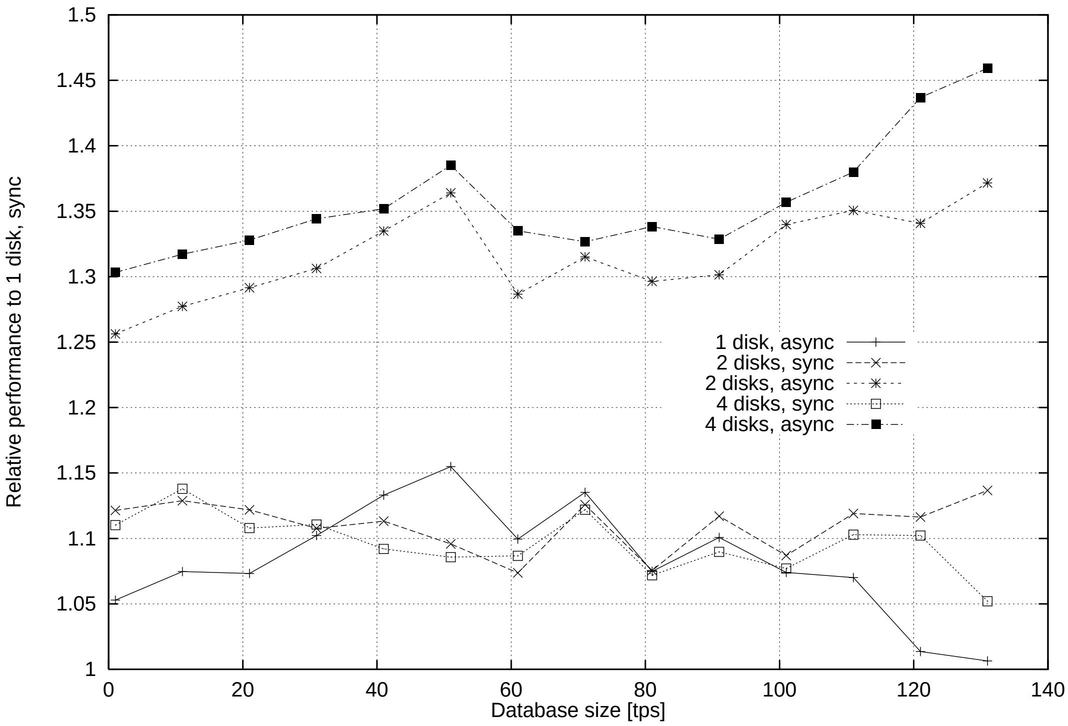


Figure 5.5: The relative effect of database residency and IO mechanisms on throughput.

garbage collection is in balance with processing the next commit group can we see a considerable speedup since that is the only opportunity when the single disk can work in parallel with the CPU.

The duration of commit groups, a rough measure of real-timeness, is shown in Figure 5.6.

We have also implemented asynchronous reading. This almost halves recovery times when using two disks, but additional disks bring only a modest further improvement.

In the previous tests we use a first generation size of 2 MB. Next we fix the disk IO method to two disks using asynchroous writing, vary the first generation size and the related mature garbage collection scheduling parameters, and observe the throughputs and latencies. The results are summarized in Figures 5.7 and 5.8.

In general, Shades uses disks in a manner very suitable for current disk technology: it issues rather long writes and seeks only a few times per commit group to write the root block, which is usually placed in the very beginning of the disk. We have also made some trials with writing the root block among the rest of the database, thereby omitting the seeks, but for reasons unknown, the benefits re-

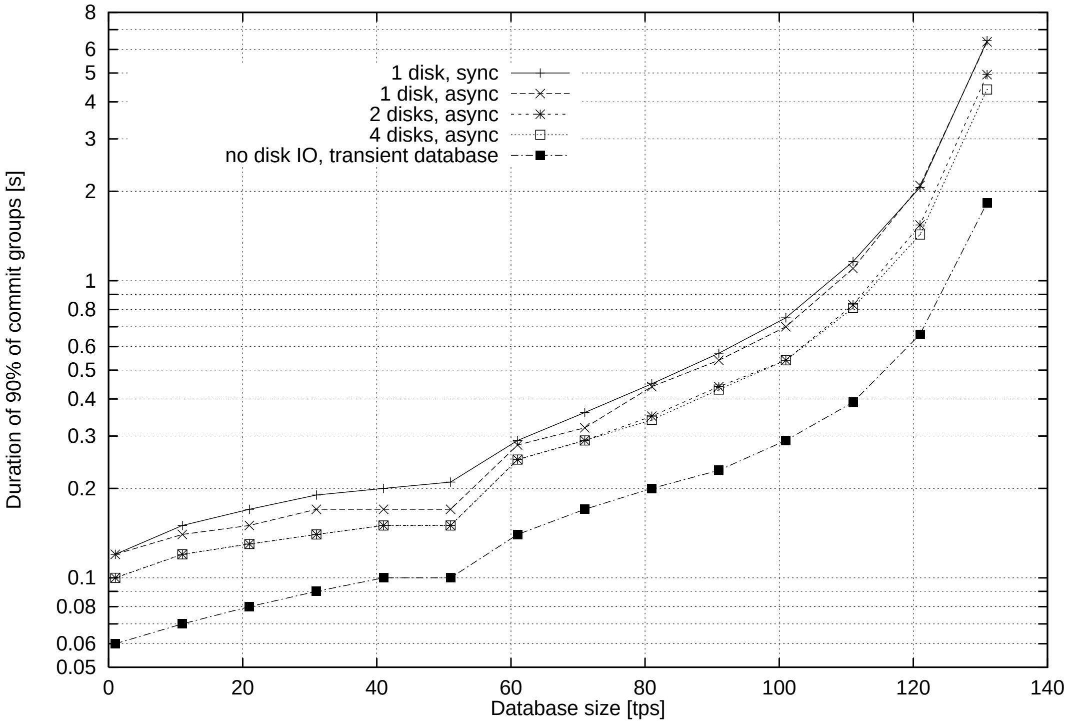


Figure 5.6: The effect of data size on real-timeness without changing the database size. 90% of commit groups finished before the time given on the y-axis.

mained negligible. During recovery the newest root block could be searched in a matter of seconds using a bisection search over the whole disk space.

Additionally we have implemented disk load balancing. While this hasn't affected our peak performance benchmarks where all disks were dedicated to the database server, disk load balancing can be quite important in practical situations where the disks are shared among other users in the same machine. Experiments on disk load balancing and optimized root block writing have been discussed more thoroughly in [51].

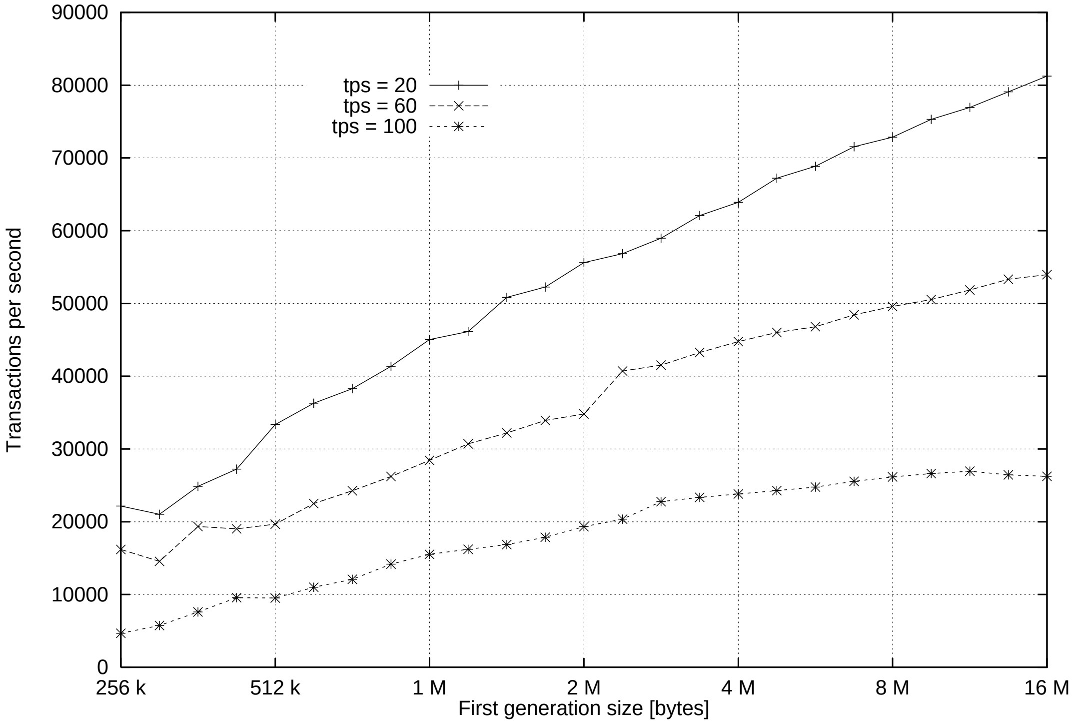


Figure 5.7: The effect of first generation size on throughput.

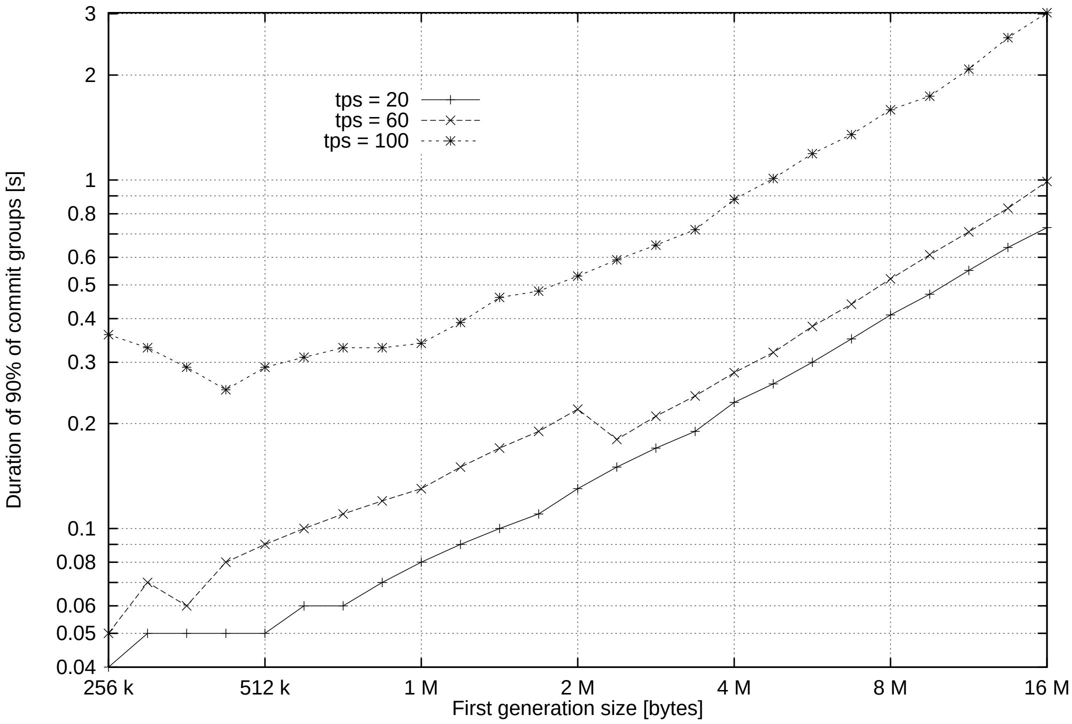


Figure 5.8: The effect of first generation size on real-timeness.

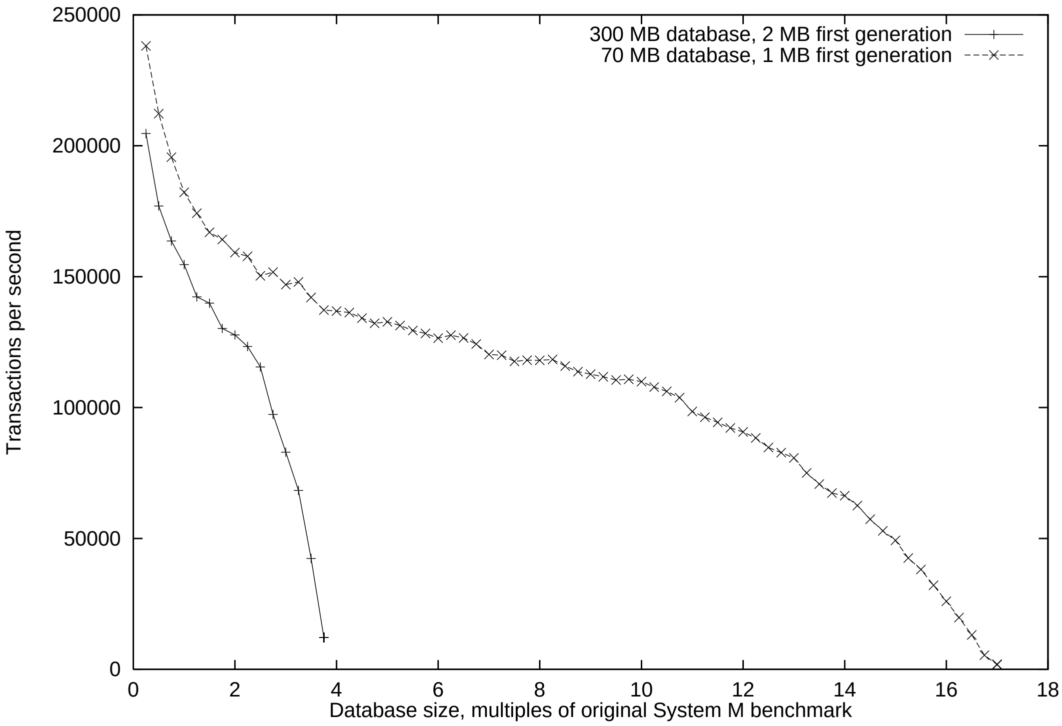


Figure 5.9: System M benchmark performance.

We shall conclude by showing in Figure 5.9 how Shades performs on the System M benchmark. System M was able to perform up to 73.1 transactions per second on an approximately one MIPS machine, a VAX 11/785 with 128 MB of main memory [77]. The 350 MHz Pentium II processor is probably approximately 500 times faster than the VAX. This would scale up to a performance of approximately 36000 transactions per second, which is still almost five times less than our 154000 transactions per second for the 70 MB database size. Furthermore, since the memory latencies and disk write performances have improved much less, perhaps by factors of four and ten, we can safely assume Shades is considerably much more performant on equivalent hardware than System M. As mentioned in the beginning of Section 5.1, we outperformed also current commercial DBMS by a huge margin in the Intelligent Network benchmark [54].

In both benchmarks, System M and Intelligent Networks, it appeared that Shades consumes *less* memory to store the same information as the relational database managers. Of course, should the information be random bits this would hardly be the case, but in practice we for example share common parts of strings, our index structures consume less memory than e.g. B-trees, and our compacting garbage collector eliminates the memory consumption overhead caused by fragmentation.

Chapter 6

Further Work

Evidently a sizable amount of work has already been put on HIBASE. Nevertheless several larger projects, numerous details, and an abundance of uninvestigated ideas remain. In this Chapter we shall cover some of them.

6.1 Shades

Shades, as described in this Thesis, has only two levels of collection: the minor collection collects the first generation and the major collection collects the whole database. However, it is common that databases contain large amounts of very static data in the oldest generations. Collecting them is wasted work if no or only little memory is freed. We could have have, for example, an intermediate collection which collects only a third of the database and runs six times more frequently than a major collection. This was actually tried in the Intelligent Network benchmark. We found it to save a considerable amount of work in several relevant cases, particularly when the first generation was less than approximately 5% of the database size and when the database was rather full. However, since recovery for additional generationality is rather complicated and has not yet been implemented, it was excluded from the benchmark results. Additional generationality also brings up numerous questions on the heuristics that control the scheduling of garbage collection and choice of the generations in the Fromspace.

Our experiments in Section 5.3 showed that remembered set sizes remain reasonable in practice. Should the contrary be found, there are algorithmic means to decrease remset sizes. The technique of duplicating frequently referred cells described in Section 2.6 seems to be the most promising approach. We have also observed that the addresses stored in large remembered sets are often rather close numerical values and easily compressible at least to 50% with simple delta-coding.

In Section 2.6 we also mentioned that while Shades is real-time in terms of disk writing per commit group, it is not necessarily real-time in terms of CPU work.

This too can be solved algorithmically, but it does require additional implementation work.

Another non-real-time primitive in the current Shades is resizing the memory area used by the mature generations. The C library function `realloc` can not be used for this purpose since it involves allocating another memory area and copying the whole database image to it, which can take a prohibitively long time and consume overly much memory. The correct solution would be to use suitable virtual memory primitives provided by some operating systems.

An interesting memory optimization that Shades could do for the application programmer is to combine equal cells. This could be done by using hash tables which memorize copied cells and their new locations. Whenever another cell is about to be copied, the hash table is first consulted whether an equivalent cell already exists in the database. The hash table for the first generation collection might contain all the cells in the generation and ensure all equal cells are indeed combined, but the cache for the whole database would necessarily have to be lossy, similar to the ones described in Section 4.2.4.2. Only after several mature generation collections would most equal cells be combined.

Maximally efficient disk IO always seems to require elaborate investigation and tweaking in the internals of disks, device drivers, and the kernel. The performance problems observed in the operating systems when using asynchronous writes are gradually being improved and will eventually be fixed by others. On our behalf, we could create even more opportunities for asynchrony by allowing even the root block to be written asynchronously. This can indeed be done, but the disk pages of the commit group must not be freed before the root block has safely hit the disk. Also disk load balancing and optimized root block writing mentioned in Section 5.4 will require some further work.

Marshalling a hierarchical data structure into a flat byte sequence and unmarshalling it back to an equivalent data structure can be performed by Shades alone. If the flat byte sequence is stored in Binary Large Objects, BLOBs, and the BLOBs are cached in free pages, HIBASE could also be used to manage efficiently mostly disk-based data. If the byte sequence is sent over a network to another server, HIBASE can be used to migrate data. Since computation (threads) is data in Shades, migration would also enable agent-based nomadic or mobile computing. While theoretically rather simple, marshalling and unmarshalling has not yet been implemented in practice.

By far the most serious project for Shades is replication. We have found algorithms to replicate the Shades database on several computers with little computational overhead, latency, and network IO. Should one replica crash or yield a different answer for example due to a random bit failure in the memory hardware, the other replicas will notice this and take over.

Numerous other smaller implementation details, such as making Shades 64-bit oriented and various micro-optimizations, will not be covered here.

6.2 Indexes

Further optimizations are always possible in the already existing index structures, but it generally appears that the gradually diminishing improvements come at an increasing complexity of implementation. Rather than developing new, slightly more efficient or concise trees, tries or queues, we will most probably focus on supporting more operations on the data structures we already have: set, cursor and range operations for tries, merging and splitting for deques, seamless support for streams, presence of weak or deteriorating pointers as mentioned in Section 2.7.3, etc. Only strings will probably have to be optimized further.

A few new data structures may be developed for dedicated keys or values. We already have tentative implementations for representing telephone numbers and tries mapping them to other data. In future we may develop maps specialized for string keys and maps specialized for boolean data.

As mentioned in Section 3.3, priority queues with merging and deletion remains an open problem, although not a very urgent one.

6.3 Shines

The performance of the byte code interpreter appears satisfactory as it is. Numerous low-level micro-optimizations will nevertheless be tried, particularly in byte code dispatching and in entering byte code sequences. Stack frames could also be mutated and reused more eagerly, but these optimizations may require additional compiler support. New byte code instructions will undoubtably be added and old ones improved. The most significant performance improvement is, however, to be expected from the Just-In-Time compiler.

Although the compiler currently produces delightfully efficient byte code, a handful of optimizations do remain. Common subexpression elimination, escape analysis, loop and recursion optimizations, and some low-level optimizations might be worth implementing when time permits or need appears.

Most effort will be put on improving Shines as it appears to the prospective programmer. Shines does not currently have floating point numbers, objects lack methods, the pattern matching and stream expressions described in Section 4.3.2.1 could be extended to be more powerful, Shines is still untyped and it has no module system except for the one provided by the C preprocessor. A book specifying the Shines language and describing the philosophy of programming in Shines is being written.

We have tentative versions of a profiler library, a toolkit for building graphical user interfaces, and a literate programming tool [41]. A debugger is currently being designed. Tools for regular expression matching, lexing, and parsing are in their early implementation stages. Support for ASN.1 may be needed in telecom applications.

Smart-pointer interfaces from Shades and the index structures to C++ and a Java have been implemented although they need to be updated. The HIBASE server can be connected through TCP/IP and in future hopefully also with Java Remote Method Invocation and interfaces specified by the Interface Definition Language of the Object Management Group. We might also build ODBC support so that HIBASE can issue SQL queries to external legacy databases.

Future research projects will involve support for schema evolution, more sophisticated type systems possibly with dependent types, byte code verification and proof carrying code, and multi-processor support.

6.4 Selling HIBASE

As mentioned in the introduction, no persistent programming language has yet succeeded commercially. We too expect to encounter considerable challenges in selling HIBASE, but there are some differences in our starting points.

Firstly, all persistent programming languages came from the academia. Most currently widely used programming languages have a business or government background: SQL and FORTRAN came from IBM, C and C++ came from AT&T, SmallTalk from Xerox, Java from Sun, COBOL was developed for businesses, Ada was mandated by U.S. Department of Defence. LISP, Pascal, and Basic were originally designed in the academia, but commercial suppliers and enhancements have been essential to their success. Some scripting languages, particularly Perl, were developed by communities of hackers, some enrolled by the academia, some by the industry.

Yet most language paradigms, programming styles, and compiler techniques originate from the acedemia. Why has the academic effort usually been insufficient for commercial success? One explanation is that academic systems are usually crude unsupported prototypes [90]. Most academics prefer theoretical research over the pragmatic and tedious tasks of fixing bugs, improving user documentation, and implementing standard programming tools and libraries of little novelty. Naturally exceptions do exist, e.g. Objective Caml may be on its way to success, but these exceptions are a clear minority.

Another explanation is that research teams in the academia are usually rather short-lived. Two years is sufficient to design and implement a programming language, but twenty years is needed to port it to new platforms, fix bugs, add features and tools, and — most importantly — support its users. After the initial effort of specifying the language, implementing the compiler, core libraries and the necessary tools, the project staffing may decrease or share their time with other tasks, but it is imperative that the users are supported and the system is regularly updated.

Unfortunately this very rarely happens in the academia since support projects are rare, exhausted graduates move to new challenges, and professors may lose interest. In other words, academic programming environments are perceived as too risky for serious application development.

Secondly, contrary to other persistent programming languages, we expect to have dozens of Shines-programmers available as soon as we are ready to invite them. Some of them will come from the industry, for example from Nokia Telecommunications, who has so generously supported the H$^{I}B_{A}$SE-project; naturally they want to see it used in practice by their programmers in developing their products. We expect some programmers to come from the functional programming community, which eagerly waits for pragmatic, industrial strength systems and has few solutions for persistence, real-timeness and replication.

The critical mass for a language is probably approximately a hundred programmers who find the language delightful and productive and use it in their daily work. Such a user community begins to have the sufficient inertia and different roles needed: grass-roots programmers, software designers and architects, pedantic language lawyers and code reviewers, teachers for other users, evangelists, workshop and special interest group organizers, and people who contribute new features, bugfixes, subroutine libraries, documentation, or merely suggestions and feedback. Judging from the success of Perl, Python, Tcl, and Objective Caml compared to some other systems, a rather free distribution policy, e.g. the General Public Licence (GPL), is highly beneficial to gain acceptance.

Thirdly, we do have a few important features our predecessors do not have: H$^{I}B_{A}$SE is real-time, probably orders of magnitude more performant, and it will be replicated in order to provide arbitrarily high reliability.

Nevertheless, we do not expect success to come easily. In addition to polishing our current capabilities and implementing new ones, we will need two things: killer applications and good timing. Java had both. The hype that brought us C++ in the beginning of the 90's had largely turned to disdain by experiences from a few generations of projects by the mid-90's. When Java was introduced, people were open for alternatives to C++ and several other languages, e.g. Smalltalk, had seemed to become more popular.

The killer application is a problem domain the programming environment solves particularly well. The killer application of Java was applets embedded in web pages. Now, a few years of hype later, Java is only rarely seen in web applets, but it has gained a strong user base in introductory programming and also seems to be slowly penetrating into some industrial domains.

No doubt the bad performance, unsupported hype, and shoddy quality of many Java tools and libraries have already gathered some negative response. Some day Java will be dethroned from its current most hyped language status, but currently only a fool would try. The killer application of H$^{I}B_{A}$SE could have been in web servers serving dynamic pages, but we were at least three years too late. New areas will definitely spring up and we will be alert, particularly if we have reached the critical user base of approximately a hundred programmers.

Chapter 7

Conclusion

We have presented the persistent memory manager Shades and the index structures and the persistent programming language, Shines, implemented on Shades. The basic premise in Shades is that updates-in-place are reserved for very limited occasions; copy-on-write is the default method of storing information in the database. The design of Shades, and consequently also the index structures and Shines, is particularly suitable for a main memory environment. Particular attention has been put on reliability, real-timeness, and high performance.

Shades combines aggressive commit grouping with real-time generational garbage collection. The first generation corresponds to the memory allocated for the processing of one commit group. When the commit group is about to commit, the first generation is garbage collected and the surviving data is placed on adjacent memory locations and written to disk.

The main challenge in persistent memory managers is to reclaim and recycle, efficiently and in real-time, the garbage located in older generations. Only a few usable approaches have been presented in the literature. They maintain various metadata in stable storage, for example remembered sets, which record the pointers that cross generation boundaries. The main novelty of Shades is that these remembered sets can be recovered after a crash and can therefore be kept in volatile memory, thereby avoiding the sizeable disk IO overhead caused otherwise.

Various index structures have been implemented and studied on Shades. Tree-like data structures can be easily mimicked from traditional text book solutions, although care and some creativity was required for maximum performance and concision. Queues, deques, and priority queues differ more fundamentally from the ones presented in text books — priority queues with merging and deletion can even be considered a theoretically open problem.

The execution mechanism of Shines, the persistent programming language, is based on a byte code interpreter whose data structures are managed by Shades. The byte code interpreter and the compiler from Shines to byte code were presented including numerous seemingly minor details, which nevertheless are significant for high performance. The scheme for bootstrapping the compiler, the CPS conversion, the interaction between the compiler and Shades for live-precise garbage collection can be regarded as considerably different from off-the-shelf solutions if not of theoretical value.

The behavior and performance of the system was studied. Suitable default values for various parameters of Shades were found for different requirements. We have demonstrated experimentally that the sizes of remembered sets remain small in Shades. We also confirmed the weak generational hypothesis for traditional programs on Shades, but in database applications we found some statistics which differ from the ones previously reported.

The resulting system has been found highly performant and in experiments it has handsomely beaten commercial relational database management systems. No one reason for the high performance of HIBASE can be given, but the main contributors are aggressive commit grouping, negligible overhead of writing metadata to disk, collecting garbage and locating the surviving data adjacently prior to writing it to disk, efficient and concise index structures, and a careful design and implementation of the entire system.

Bibliography

[1] S. Abdullahi, E. E. Miranda, and G. Ringwood. Distributed garbage collection. In Bekkers and Cohen [10].

[2] O. Agesen, D. Detlefs, and J. E. B. Moss. Garbage collection and local variable type-precision and liveness in Java Virtual Machines. *ACM SIGPLAN Notices*, 33(5):269–279, May 1998.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.

[4] G. Akerholt, K. Hammond, S. P. Jones, and P. Trinder. A parallel functional database on GRIP. In R. Heldal, C. K. Holst, and P. L. Wadler, editors, *Functional Programming, Glasgow 1991: Proceedings of the 1991 Workshop, Portree, GB*, Lecture Notes in Computer Science, pages 1–24, Berlin, DE, 1992. Springer-Verlag. ISBN : 3-540-19760-5.

[5] A. W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, 1989.

[6] A. W. Appel. A runtime system. *Lisp and Symbolic Computation*, 3:343–380, 1990.

[7] A. W. Appel. *Compiling with Continuations*, chapter 16, pages 205–214. Cambridge University Press, 1992.

[8] S. Ashwin, P. Roy, S. Seshadri, A. Silberschatz, and S. Sudarshan. Garbage collection in object oriented databases using transactional cyclic reference counting. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 366–375, 1997.

[9] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, Nov. 1983. Also published in "Readings in Object-Oriented Database Systems" edited by S. Zdonik and D. Maier, Morgan Kaufman, 1990.

[10] Y. Bekkers and J. Cohen, editors. *Proceedings of International Workshop on Memory Management*, volume 637 of *Lecture Notes in Computer Science*, St Malo, France, 16–18 Sept. 1992. Springer-Verlag.

[11] R. Bird, G. Jones, and O. de Moor. More haste, less speed: lazy versus eager evaluation. *Journal of Functional Programming*, 7(5):541–547, Sept. 1997.

[12] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 25–37, San Diego, California, 19–21 Jan. 1998.

[13] G. S. Brodal and C. Okasaki. Optimal purely functional priority queues. *Journal of Functional Programming*, 6(6), Dec. 1996.

[14] S. Chakravarthy. Architecture and monitoring techniques for active database: An evaluation. Technical Report UF-CIS-TR-92-041, University of Florida, Department of Computer and Information Sciences, Nov. 1992.

[15] C. J. Cheney. A non-recursive list compacting algorithm. *Communications of the ACM*, 13(11):677–8, Nov. 1970.

[16] T.-R. Chuang and B. Goldberg. Real-time deques, multihead turing machines, and purely functional programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 289–298, New York, NY, USA, June 1993. ACM Press.

[17] S. Dar and R. Agrawal. Extending SQL with generalized transitive closure. *IEEE Transactions on Data and Knowledge Engineering*, 5(5):799–812, Oct. 1993.

[18] A. Diwan, D. Tarditi, and J. E. B. Moss. Memory subsystem performance of programs using copying garbage collection. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices. ACM Press, Jan. 1994.

[19] J. K. Doyle, J. E. B. Moss, and A. L. Hosking. When are bytecodes faster than direct execution? Unpublished manuscript, 1997.

[20] J. R. Driscoll, D. D. K. Sleator, and R. E. Tarjan. Fully persistent lists with catenation. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 89–99, San Francisco, California, 28–30 Jan. 1991.

[21] J. R. Ellis, K. Li, and A. W. Appel. Real-time concurrent collection on stock multiprocessors. Technical Report DEC–SRC–TR–25, DEC Systems Research Center, Palo Alto, CA, Feb. 1988.

[22] M. A. Ertl. Stack caching for interpreters. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 315–327, 1995.

[23] R. R. Fenichel and J. C. Yochelson. A Lisp garbage collector for virtual memory computer systems. *Communications of the ACM*, 12(11):611–612, Nov. 1969.

[24] E. Fredkin. Trie memory. *Communications of the ACM*, 3(9):490–499, Sept. 1960.

[25] H. Garcia-Molina and K. Salem. Main memory database systems: An overview. *IEEE Transactions on Data and Knowledge Engineering*, 4(6):509–516, Dec. 1992.

[26] A. Gawecki, F. Matthes, J. W. Schmidt, and S. Stamer. Persistent object systems: From technology to market. In M. Jarke, editor, *27. Jahrestagung der Gesellschaft für Informatik*. Springer-Verlag, Sept. 1997.

[27] M. J. R. Gonçalves. *Cache Performance of Programs with Intensive Heap Allocation and Generational Garbage Collection.* PhD Thesis, Princeton University, May 1995.

[28] M. J. R. Gonçalves and A. W. Appel. Cache performance of fast-allocating programs. In *Record of the 1995 Conference on Functional Programming and Computer Architecture*, June 1995.

[29] G. H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures.* Addison-Wesley, New York, 2. edition, 1991.

[30] J. Gray, editor. *The Benchmark Handbook for Database and Transaction Processing Systems.* Morgan Kaufmann Publishers, San Francisco, 2. edition, 1993.

[31] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. a. Price, F. Putzolu, and I. Traiger. The recovery manager of the System R database manager. *Computing Surveys*, 13(2):223–242, June 1981.

[32] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann Publishers, San Mateo (CA), USA, 1993.

[33] D. Grunwald, B. Zorn, and R. Henderson. Improving the cache locality of memory allocation. In PLDI [73], pages 177–186.

[34] D. Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, University of Arizona, Department of Computer Science, Tucson, Arizona, 1993.

[35] R. Hood and R. Melville. Real-time queue operation in pure LISP. *Information Processing Letters*, 13(2):50–54, 13 Nov. 1981.

[36] R. L. Hudson and J. E. B. Moss. Incremental garbage collection for mature objects. In Bekkers and Cohen [10].

[37] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. Technical Report COINS 91-47, University of Massachusetts at Amherst, Dept. of Computer and Information Science, Sept. 1991.

[38] J.-P. Iivonen. Menetelmä FIFO-jonon toteuttamiseksi muistissa ja muistijärjestely, Sept. 1998. Finnish patent application nr. 98191.

[39] J.-P. Iivonen, S. Nilsson, and M. Tikkanen. An experimental study of compression methods for functional tries. Submitted to WAAPL'99, 1999.

[40] U. Jaeger and J. C. Freytag. An annotated bibliography on active databases. *SIGMOD*, 24(1):58–69, March 1995.

[41] C. Jing. NolangWEB — a literate programming tool for Nolang. Master's Thesis, Helsinki University of Technology, May 1999.

[42] R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, July 1996. With a chapter on Distributed Garbage Collection by R. Lins. Reprinted February 1997.

[43] S. P. Jones, W. Partain, and A. Santos. Let-floating: moving bindings to give faster programs. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 1–12, Philadelphia, Pennsylvania, 24–26 May 1996.

[44] B. Kao and H. Garcia-Molina. *An Overview of Real-Time Database Systems*, chapter 19. Prentice Hall, 1995.

[45] H. Kaplan and R. E. Tarjan. Persistent lists with catenation via recursive slowdown. In *Proceedings of the Twenty-Seventh Annual ACM Symposium on the Theory of Computing*, pages 93–102, Las Vegas, Nevada, 29 May–1 June 1995.

[46] J. Kent and H. Garcia-Molina. Optimizing shadow recovery algorithms. *IEEE Transactions on Software Engineering*, 14(2):155–168, Feb. 1988.

[47] D. J. King. Functional binomial queues. In K. Hammond, D. N. Turner, and P. M. Sansom, editors, *Glasgow functional programming workshop*, Ayr, Scotland, 1994. Springer-Verlag.

[48] G. N. C. Kirby, R. Morrison, R. C. H. Connor, and S. B. Zdonik. Evolving database systems: A persistent view. Technical Report CS/97/5, University of St Andrews, 1997.

[49] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery for a large stable heap. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):177–186, June 1993.

[50] E. K. Kolodner and W. E. Weihl. Atomic incremental garbage collection and recovery for large stable heap. In P. Buneman and S. Jajodia, editors, *Proceedings of 1993 ASM SIGMOD International Conference on the Management of Data*, pages 177–186, Washington, DC, May 1993. Also MIT/LCS/TR-534, February, 1992.

[51] A.-P. Liedes. Tik-76.139 operating systems course assignment, 1999.

[52] U. Maheshwari and B. Liskov. Collecting cyclic distributed garbage by controlled migration. In *Proceedings of PODC'95 Principles of Distributed Computing*, 1995. Later appeared in Distributed Computing, Springer Verlag, 1996.

[53] U. Maheshwari and B. Liskov. Partitioned garbage collection of a large object store. In *Proceedings of SIGMOD'97*, 1997.

[54] E. Mäkelä, P. Nuuttila, M. Tikkanen, P. Pulkkinen, T. Pystynen, and C. Worton. Nokia IN OODBMS/RDBMS benchmark test report, March 1998. Nokia Telecommunications Report NTCDIGFD203/02.EN.

[55] L. Malmi. On updating and balancing relaxed balanced search trees in main memory. Research Report TKO-A-35, Laboratory of Information Processing Science, Helsinki University of Technology, 1997. D.Tech. Thesis.

[56] M. Mannino., I. Choi, and D. Batory. The object-oriented functional data language. *IEEE Transactions on Software Engineering*, 16(11):1258–1272, Nov. 1990.

[57] B. Mathiske, F. Matthes, and J. W. Schmidt. On migrating threads. *Journal of Intelligent Information Systems*, 8(2):167–191, 1997.

[58] F. Matthes and J. W. Schmidt. Persistent threads. In *Proceedings of the Twentieth International Conference on Very Large Data Bases, VLDB*, pages 403–414, Santiago, Chile, Sept. 1994.

[59] F. Matthes, G. Schröder, and J. W. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M. P. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.

[60] D. J. McNally, S. Joosten, and A. J. T. Davie. Persistent functional programming. In *Fourth Int'l Workshop on Persistent Object Sys.*, page 59, Martha's Vineyard, MA, Sept. 1990.

[61] R. Morrison, R. C. H. Connor, Q. I. Cutts, G. N. C. Kirby, D. S. Munro, and M. P. Atkinson. The napier88 persistent programming language and environment. *The FIDE Book*, 1999.

[62] R. Morrison, R. C. H. Connor, G. N. C. Kirby, and D. S. Munro. Can java persist? In *1st International Workshop on Persistence for Java*, Glasgow, 1996.

[63] J. E. B. Moss, D. S. Munro, and R. L. Hudson. PMOS: A complete and coarse-grained incremental garbage collector for persistent object stores. In *Proceedings of the 7th International Workshop on Persistent Object Systems*, pages 140–150. Morgan Kaufmann, 1996.

[64] D. S. Munro, A. L. Brown, R. Morrison, and J. E. B. Moss. Incremental garbage collection of a persistent object store using PMOS. In *Eigth International Workshop on Persistent Object Systems*, Tiburon, California, 1998.

[65] S. M. Nettles and J. W. O'Toole. Real-time replication-based garbage collection. In PLDI [73].

[66] C. Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[67] S. Paajanen. Tietorakenteiden tietoa tuhoamaton toteutus. Master's Thesis, University of Helsinki, May 1998.

[68] N. Paton, R. Cooper, H. Williams, and P. Trinder. *Database Programming Languages.* International Series in Computer Science. Prentice Hall, 1996.

[69] M. Pettersson. A term pattern-match compiler inspired by finite automata theory. In U. Kastens and P. Pfahler, editors, *Proceedings of the Fourth International Conference on Compiler Construction*, number 641 in Lecture Notes in Computer Science, pages 258–270. Springer-Verlag, 1992.

[70] S. L. Peyton Jones and D. R. Lester. *Implementing Functional Languages: A Tutorial.* Prentice-Hall, Hemel Hempstead, 1992.

[71] N. Pippenger. Pure versus impure LISP. In *Conference Record of POPL '96: The* 23rd *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 104–109, St. Petersburg Beach, Florida, 21–24 Jan. 1996.

[72] I. Piumarta and F. Riccardi. Optimizing direct-threaded code by selective inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-98)*, volume 33,5, pages 291–300, New York, June 17–19 1998. ACM Press.

[73] *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Albuquerque, New Mexico, June 1993. ACM Press.

[74] M. B. Reinhold. *Cache Performance of Garbage-Collected Programming Languages.* PhD Thesis, MIT Laboratory for Computer Science, Sept. 1993. Also Technical Memo MIT/LCS/TR–581.

[75] M. B. Reinhold. Cache performance of garbage-collected programs. In *Proceedings of SIGPLAN'94 Conference on Programming Languages Design and Implementation*, volume 29 of *ACM SIGPLAN Notices*, Orlando, Florida, June 1994. ACM Press. Also Lisp Pointers VIII 3, July–September 1994.

[76] M. Rossi and K. Sivalingam. A survey of instruction dispatch techniques for byte code interpreters. In K. Oksanen, editor, *Seminar on Mobile Code*, number TKO-C-79, Laboratory of Information Processing Science, Helsinki University of Technology, 1995.

[77] K. Salem and H. Garcia-Molina. System M: A transaction processing testbed for memory resident data. *IEEE Transactions on Data and Knowledge Engineering*, 2(1):161–172, Mar 1990.

[78] P. M. Sansom and S. L. Peyton Jones. Generational garbage collection for Haskell. In R. J. M. Hughes, editor, *Record of the 1993 Conference on Functional Programming and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, University of Glasgow, June 1993. Springer-Verlag.

[79] N. Sarnak and R. E. Tarjan. Planar point location using persistent search trees. *Communications of the ACM*, 29(7):669–679, July 1986.

[80] J. Seligmann and S. Grarup. Incremental mature garbage collection using the train algorithm. In O. Nierstras, editor, *Proceedings of 1995 European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, University of Aarhus, Aug. 1995. Springer-Verlag.

[81] M. Serrano. Inline expansion: when and how? In *Proceedings of the conference on Programming Languages, Implementation and Logic Programming.*, page 15, Shouthampton, Sept. 1997.

[82] D. Sitaram. Handling control. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation: Albuquerque, New Mexico, June 23–25, 1993*, SIGPLAN notices; ISSN: 0362-1340; v. 28, no. 6 (June 1993), pages 147–155, New York, NY 10036, USA, 1993. ACM Press.

[83] M. Skubiszewski and P. Valduriez. Concurrent garbage collection in O2. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 356–365, 1997.

[84] C. Small. A functional approach to database updates. *Information Systems*, 18(8):581–595, Dec. 1993.

[85] R. M. Stallman. *Using an Porting GNU CC*. Free Software Foundation, Inc., 1.39 edition, Jan. 1991.

[86] J. Stankovic and W. Zhao. On real-time transactions. *ACM, SIGMOD Record*, 17(1):4–18, 1988.

[87] P. Trinder. A functional database. Research Report CSC/90/R10, Department of Computing Science, University of Glasgow, Glasgow, UK, 1990. D.Phil. Thesis.

[88] D. M. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, Apr. 1984. Also published as ACM Software Engineering Notes 9, 3 (May 1984) — Proceedings of the ACM/SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, pp. 157–167, April 1984.

[89] S. Vegsal and U. F. Pleban. The runtime environment of Screme, a Scheme implementation for the 88000. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS III)*, pages 172–182, Boston, Apr. 1989.

[90] P. Wadler. Functional programming: Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, August 1998. Functional programming column.

[91] P. R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan. 1994.

[92] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching consideration for generational garbage collection: A case study of large and set-associative caches. Technical Report UIC–EECS–90–5, University of Illinois at Chicago EECS Dept., Chicago, Illinois, Dec. 1990. Improved version appears in [**?**].

[93] P. R. Wilson, M. S. Lam, and T. G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on Lisp and Functional Programming*, pages 32–42, San Francisco, CA, June 1992. ACM Press.

[94] T. Ylönen. *Shadow Paging is Feasible.* Licentiate's Thesis, Helsinki University of Technology, Department of Computer Science, May 1994.

[95] B. Zorn. The effect of garbage collection on cache performance. Technical Report CU–CS–528–91, University of Colorado at Boulder, Department of Computer Science, Boulder, Colorado, May 1991.

Index