Thesis

# A Swap Dataspace Manager for Drops

Sebastian Sumpf

12th March 2006

University of Technology Dresden
Department of Computer Science
Institute of System Architecture
Chair of Operating Systems

| | |
|---|---|
| Professor: | Prof. Dr. rer. nat. Hermann Härtig |
| Tutor: | Dipl.-Inf. Ronald Aigner |

## Statement

Hereby I declare, that this work was produced in an autonomous fashion (except chapter “State of the Art” which was inspired by a whole lot of papers) and that I did use the specified resources, only.

12th March 2006, Dresden

Sebastian Sumpf

## Acknowledgenements

**“Thanks Mike!”**

# Contents

# List of Figures

# 1 Introduction

One of L4's virtual-memory system (VM) goals is to provide VM *diversity*: "Applications should be able to build and customize the VM according to their needs. They should have complete control over the VM policies ..." [ALE$^+$00]. In practice developers often settle for a policy in kernel that comes closest to the requirements of most applications.

To be able to describe this project's role in the L4 environment, some essential concepts will be briefly examined next.

## 1.1 L4

L4 is a second generation microkernel interface developed by Jochen Liedtke. While the first generation of microkernels (like Mach) suffered from significant performance losses, in comparison to Unix, Liedtke improved performance by a thinner IPC (inter-process communication) layer and other simplifications of the original kernel design, leading to a more "real-world" solution.

In general microkernels were an attempt to break out of the ever-growing operating system kernels at the time. The reason for the vast growing of os-kernels was mainly caused by the support of more and more devices, leading to more and more kernel-driver code. Microkernels choose the opposite direction by offering a very simple abstraction over the hardware, with a set of primitives that implement minimum operating system services, thus aiming to build flexible, small, and modular operating systems[1]. Many of the services a (from the microkernel point of view) *monolithic* kernel offers are implemented in *user-space* by so called *servers* communicating through IPCs.

The kernel this paper deals with is called *Fiasco* and was developed at this very same institute. *Fiasco* is an implementation of the L4 interface, especially aiming to provide real-time support as well as, enhancements of the security in operating systems. All the user-space servers and outside-kernel features form the "Dresden Real-time Operation System", referred to as DROPS.

## 1.2 L4 - abstractions

L4 provides two abstractions: *threads* and *address-spaces*. The thread is a unit of execution, it is associated to a unique address-space, and executes user-space code. Threads can communicate witch each other using IPC operations provided by the microkernel. An address-space and the thread(s) associated to it are referred to as a L4 task. IPCs in

[1] Note that even Linux now features a modularized approach

L4 are synchronous, unbuffered, and can be used within and across tasks. An address-space defines the virtual memory of the threads associated with it.

## 1.3 Virtual memory management

For virtual memory management[2] L4 supports the notion of per-thread *pagers* to be able to handle page-faults. A pager in L4 is a thread that is invoked by the kernel if a page-fault occurs. Every L4-thread has a (potentially different) pager assigned. On page-fault the kernel sets up an IPC to the pager, specifying the faulting thread as the sender.

Also, the L4-kernel permits hierarchical pagers, which implies that a pager's address-space might itself be managed by another pager. For such a hierarchical scheme to work, an initial address-space is needed, which is occupied by the first task (or first pager). This address-space is called $\sigma_0$ and the DROPS task is (obviously) called "sigma0". All the available physical memory is "granted" (see below) to this task during system startup.

Following address-space management operations are provided by the kernel:

**Grant:** If a thread *grants* any of its pages to another thread, this page is removed from the granter's address-space and included in the grantee's address-space. Of course the granter can only grant pages it already owns.

**Map:** The *map* operation is similar to the *grant* operation, with the difference that the mapped page remains accessible in both address-spaces, and is not removed from the mapper's address-space. The mapper still "owns" the page.

**Unmap:** A thread can *unmap* a page, that it mapped to other address-spaces. In DROPS the thread can choose if the page is unmapped in other address-spaces only, or in its own as well. The next reference of a thread to an unmapped page will result in a page-fault.

Hierarchical pagers allow the stacking of pagers, e.g. one pager can be build on top of another, extending or changing the functionality of the underlying pager. The actual realization of the pager concept in DROPS is discussed next.

### 1.3.1 The dataspace concept

L4's virtual-memory concept is based upon dataspaces. A dataspace is an unstructured data container, thus an entity that contains data. Example data for dataspaces are pages, frame buffers, device blocks .... Dataspaces can be *attached* to *regions* of an address-space. When accessing the virtual memory of a region, one effectively accesses the dataspace of that region. As a result of this strategy, one has to make sure that only one pager manages a specific address-region, to be able to prevent possible conflicts (for example concurrent accesses), thus inducing the need for a memory management

[2] This section closely follows [ALE+00] as well as the lectures "Microkernel based systems" and "Microkernel construction" of TU-Dresden.

which is independent of pagers. The solution proposed by L4 is called *region-mapper*. A region-mapper keeps track of what part of the (virtual-)address-space is managed by which pager. It can be seen as a 3-tuple $(region - start, region - end, pager)$. Every page-fault is caught by the region-mapper. It translates the faulting address and forwards the page-fault (IPC) to the pager that corresponds to the the given address. In Drops every task has its own region-mapper thread (l4rm).

The actual pagers are called *dataspace managers* in L4 and they are implementing dataspaces. Each dataspace manager has the opportunity to support different semantics. For instance, one dataspace manager might offer physical memory, another one paged memory using backup storage, a third one copy one write pages, a fourth one Unix files, and so on. Usually, dataspace managers cache the contents of dataspaces in their own virtual address-space and use the microkernel operations to satisfy page faults. Note that dataspaces are a higher level concept and that the kernel is unaware of dataspaces.

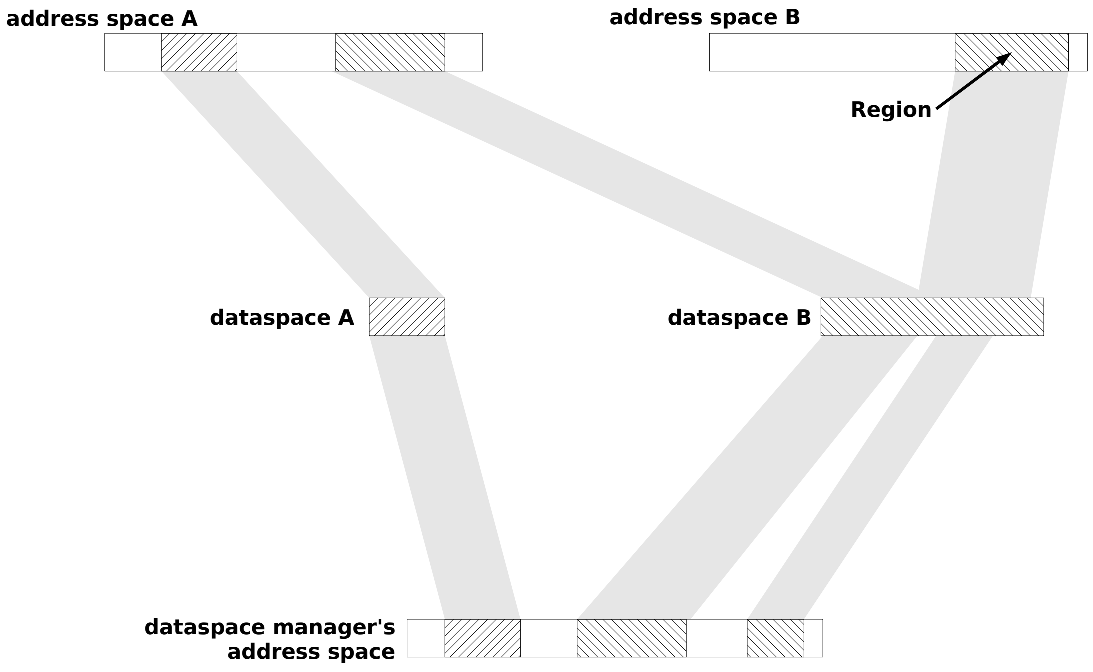

***Figure 1.1:*** Relationship between address-spaces, dataspaces, and regions. Address-space *A* has two regions, to which dataspaces *A* and *B* are attached. Dataspace *B* is additionally attached to a region of address-space *B*.

Sequence of steps from an application's memory request to memory access:

1. An application requests memory. A dataspace is created and attached to some virtual memory region.

2. The application accesses a page in the freshly attached region, this leads to a page-fault.
3. The kernel forwards the page-fault to the region-mapper through IPC.
4. The region mapper translates the faulting address into a dataspace fault and sends information (dataspace identification and offset) to the appropriate dataspace manager.
5. The dataspace manager allocates (caches) the contents of the requested page in a mapped page of its own virtual address-space.
6. The virtual page is mapped into the application's address-space.

### 1.3.2 Flexpages

L4 defines a vm-page-concept called *flexpages*. As the name suggests, this concept describes the support of "flexible" page sizes by the kernel. The only restriction that applies is that page sizes must be a multiple of the actual hardware page size. *Fiasco* in its L4v2 implementation currently supports page sizes of 4KB and 4MB (a.k.a. superpages) on the x86-architecture.

## 1.4 The swappable-dataspace manager

The goal of this project is to provide a dataspace manager for DROPS using the concepts described above. DROPS already offers a dataspace manager called DMPhys. DMPhys provides physical memory directly mapped from $\sigma_0$. The swappable dataspace manager will be build on top of DMPhys and will extent DMPhys's functionality by the possibility to move pages to some backup storage device (i.e. replace them) and vice versa.

As a consequence a page replacement policy has to be chosen. There have been some new developments in the field of page-replacement strategies during the past few years. A couple of the newer page-replacement algorithms will be described next. Afterwards block device support in DROPS is considered. Chapter 3 is concerned with how the swappable dataspace manager can be integrated into DROPS and what replacement strategy is appropriate, while chapter 4 describes actual implementation details. The last chapter (5) gives a basic performance and memory overhead evaluation.

# 2 State of the art

## 2.1 The rise from the dead

The basic assumptions behind LRU (Least recently used) have been invalidated by streaming media, garbage collection and other things possible in large address spaces, but until 2002 there weren't many replacements available that where suitable for implementation in general purpose operation systems.

**Some advantages of LRU:**

1. Simple to implement
2. Constant time and space overhead
3. Captures "recency" which is common to many workloads

**Some disadvantages of LRU:**

1. On every cache hit, the page must be moved to the most recently used (MRU) position. This action has to be protected by a lock in an asynchronous thread environment
2. In a virtual memory setting the overhead of moving a page to the MRU position – on every page hit – is unacceptable
3. LRU captures the "recency" of pages, it does not capture the "frequency" features of a workload. More generally, if some pages are requested often, but the cache size is larger then the temporal distance between two consecutive cache hits
4. LRU can be easily polluted by a scan e.g. by a sequence of one-time-only pages (i.e. not scan-resistant)
5. LRU tends to remove cold pages; if the faulting page is cold however, then LRU may replace a cold one for a warmer one. The cold page will reside in the cache for a considerable amount of time, too .
6. LRU doesn't resolve the *Correlated reference problem*, emerging whenever a page receives many references over a short period of time and than no reference for a long period of time

The newer generation of replacement algorithms tries to solve one or more of the above disadvantages. Examples are 2Q, LIRS, ARC and CAR, which will be briefly and chronologically introduced next. Common and new to all of them is the tendency to distribute pages among more than one set.

### 2.1.1 2Q(ueues)

This algorithm dates back to 1994 and is the first one I could find, that arranges page information using two sets (queues). 2Q claims to provide constant time overhead and requires no tuning. In its full version 2Q aims to improve LRU on disadvantages 2, 5 and 6. In its simple version 2Q uses two queues, $Am$ which is the main buffer and managed in LRU fashion. $A1$ is a special queue managed in FIFO order. 2Q mainly tries to only admit hot pages to the cache at all times.

***Policy***

New pages are admitted to the $A1$ queue only. Hits in $Am$ are treated like LRU and the page is moved to the front of the queue. Hits in $A1$ put the page into the front of $Am$. Page replacement is performed if some page enters $A1$ and $A1$ exceeds a certain (**tunable**) threshold. If, on the other hand, $A1$ is below this threshold and there are no free pages available, then the LRU page of $Am$ is evicted (*Note:* The new page is still placed into $A1$).

Since the simple version is hard to tune (size of $A1$), and 2Q performed poorly under real access patterns (especially *Correlated references*), the full version divides $A1$ into $A1_{in}$ and $A1_{out}$. For both new size parameters $K_{in}$ and $K_{out}$ are introduced.

Newly referenced pages are now put in $A1_{in}$ to satisfy repeated requests. A referenced page in $A1_{in}$ is not promoted to $Am$, since the access may be a correlated reference. The $A1_{out}$ buffer is used to detect pages that have high long term access rates. For a detailed description of the algorithm please refer to [JS94].

Since $K_{in}$ and $K_{out}$ still require some tuning, the values of 25% of the page slots for $K_{in}$ and 50% for $K_{out}$ are claimed to be sufficient.

### 2.1.2 LIRS – Low Inter-reference Recency Set

Another low overhead replacement algorithm was originally presented by the college of Williamsburg in 2002 [JZ02]. It attempts to improve upon 2Q and other algorithms that use additional history information besides recency, and tries to decrease complexity. LRU disadvantages 3, 4, 5 and 6 are reprocessed. Again page information is divided into two sets called High Inter-reference Recency (HIR) and Low Inter-reference Recency (LIR). Each page in the cache resides either in HIR or in LIR, dividing the cache size $L$ in $L_{hir}$ and $L_{lir}$ where $L = L_{hir} + L_{lir}$.

***Policy***

*Definition:* **Recency** refers to the number of other pages accessed between two consecutive references to the observed page. This especially refers to the number of pages accessed from the last reference to the current time.

*Definition:* Inter-reference Recency or **IRR** is defined as the **recency** between the last and the second-to-last reference of a page. It is infinite if this page hasn't been observed yet (new page or freshly swapped-in page).

The LIR set holds pages with a low IRR value, while the HIR set keeps pages with a high IRR value.

*Assumption:*
If the IRR of a page is large, the next IRR of a page is likely to be large again. In consequence pages with large IRRs are selected for replacement, and only pages in the HIR set are evicted.

The above statement implies that all pages in the LIR set must fit into the cache. Only a very small portion of the cache is assigned to the HIR set (the paper [JZ02] states that this value is about 1% of the cache size), so pages in the HIR set may be present (**resident**) or on backup storage (**non-resident**). If the recency of a page in LIR increases up to a certain point and a HIR block gets accessed at a smaller recency, the two pages are switched. The previous LIR block ends up in HIR and will be, due to the small HIR size, evicted in the near future, while the formally HIR page will stay in cache without misses as long as it resides in the LIR set. This is called Low Inter-reference Recency (LIRS) because the LIR set is what the algorithm tries to identify and keep in cache. It is stated that LIRS can be implemented using an LRU stack, thus reducing implementation overhead.

### Example

Since LIRS can be hardly explained using words only, a short example[1]is given.

| | | | | | | | | | | | Recency | IRR |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| E | | | | | | | | | x | | 0 | inf |
| D | | x | | | | | x | | | | 2 | 3 |
| C | | | | x | | | | | | | 4 | inf |
| B | | | x | | x | | | | | | 3 | 1 |
| A | x | | | | | x | | x | | | 1 | 1 |
| Pages / Virtual Time | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | | |

***Table 2.1:*** Example of how a victim block is selected. An ‘x’ means that the page has been referenced. Recency and IRR represent the corresponding values at time 10. Assumptions $L_{lirs} = 2$ and $L_{hirs} = 1$. At time 10 the LIR set $= \{A, B\}$, HIR set $= \{C, D, E\}$, the only present (resident) HIR page is E.

The recency value of page E at time 10 is 0, because no other page has been referenced since the last reference of E (at time 9), its IRR value is infinite (per definition of IRR) for E has only been referenced once. In contrast the recency of page B is 3. This is because pages A, D and E had hits since B’s last reference. Between the first and second reference of B (times 3 and 5), only page C was seen (at time 4), resulting in an IRR value of 4. E is the only present (resident) page in $L_{hirs}$ because it is the most recent one (*note:* $|L_{hirs}| = 1$).

[1] As presented in [JZ02], pages 4–5

*Assumption:* Page D is referenced at time 10. In consequence a miss occurs. The LIRS algorithm evicts page E (LRU resident of $L_{hirs}$), instead of B which would have been chosen by LRU. The new IRR of D becomes 2, which is smaller than the recency of page B(= 3). Since the IRR of D is smaller than that of B, a status switch is initiated. D enters the LIRS set while B is moved to the HIR set and will probably be evicted soon.

*Assumption:* Page C is referenced at time 10. Its IRR will be 4, which is larger than any other recency in LIR. So E is replaced by C and enters the HIR set.

### 2.1.3 ARC – A self-tuning, Replacement Cache

Developed by IBM and presented at the USENIX conference in 2003, ARC [MM03] is one of the latest developments regarding replacement policies. ARC solves LRU disadvantages 3, 4, and 5, is a real self-tuning algorithm and uses history information of more pages than actually fit into the cache.

Suppose a cache can hold $c$ pages, then ARC manages a cache directory of $2c$ pages. The cache directory is divided into two lists; $L_1$, which holds pages that have been seen only once recently, and $L_2$, that keeps track of pages which have been seen at least twice recently.

***Policy***[2]

Replace page in $L_1$ if there are exactly $c$ pages in $L_1$, replace in $L_2$ otherwise.

The actual ARC expands the above algorithm by dividing $L_1$ and $L_2$ into a top and a bottom half ($T_1$, $B_1$, $T_2$, $B_2$). Pages in $T_1$ ($T_2$) are more recent than pages in $B_1$ ($B_2$), where $|T_1 \cup T_2| = c$, if $|L_1 \cup L_2| >= c$. The algorithm defines a *target size* for $T_1$.

***ARC Policy***

Replace page in $T_1$ if $T_1$ contains at least *target size* pages; replace in $T_2$ otherwise.

On replacement in $T_1$, the replaced page is moved to the MRU position of $B_1$, $T_2$ replacement works in the same way, the replaced page is moved respectively to $B_2$. The target size varies (*adapts*) according to the workload.

***Adaption Rule***

Increase *target size* if a hit in the history list $B_1$ is registered; decrease *target size* on a hit in $B_2$ (and move page to $T_1$ (resp. $T_2$)).

This way ARC quickly adapts to variety of workloads and tunes itself completely online. The IBM measured overhead is about 0.75% of the cache size and has constant time implementation complexity.

[2] This policy is called DBL($2c$).
*Authors note:* Haven't found a resource that specifies what DBL stands for, yet.

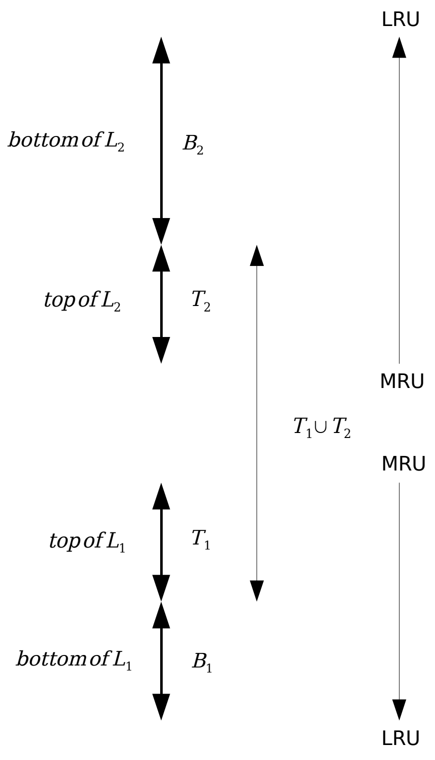

***Figure 2.1:*** General structure of the generic cache replacement policy provided by ARC. The pages in $T_1 \cup T_2$ are located in the managed cache.

### 2.1.4 CAR – Clock with Adaptive Replacement

Inspired by ARC, the CAR (2004, [BM04]) algorithm goes even further. It tries to approximate ARC in a clock-like fashion [Cor69], improving on LRU disadvantages 1, 2, 3, 4 and 5. Even the *Correlated Reference problem* 6 can be solved using a modified version of the CAR algorithm known as CART (CAR with temporal reference).

The core of CAR works (almost) in same way as ARC does. One of the main goals of CAR is to remove the need of queue reordering on every cache hit of a page hit, giving better performance on large address spaces (e.g. virtual memory settings). Again CAR uses the four lists introduced by ARC ($T_1, T_2, B_1, B_2$).

#### *Differences between ARC*

The reordering of cache hits in $T_1, T_2$, as well as the promotion of pages form $T_1$ to the $T_2$ list is executed on page replacement only. For this to work $T_1$ is (logically) divided into $T_1$ (all pages which have **not** been referenced lately) and $T_1^{'}$ (all pages that have been referenced lately). The ARC target size is now applied to $T_1 \backslash T_1^{'}$ (not $T_1$).

***Policy***
If there are *target size* pages in $T_1 \setminus T_1^{'}$, replace page in $T_1 \setminus T_1^{'}$; otherwise replace in $T_2 \cup T_1^{'}$.

For replacement in $T_1$ the list is searched (starting from the LRU position), if a page in $T_1$ has its reference bit set, reset it (set to zero), move page to MRU position in $T_2$ ($T_2 \cup T_1^{'}$), and continue with next page in $T_1$. If a page is found where the reference bit is not set (is zero), remove page from cache and move it to the MRU position of $B_1$. Replacement in $T_2$ is slightly different. Searching starts again at the LRU position, if a page with its referenced bit set is found, clear bit and make it MRU page of $T_2$. If a page with unset reference bit is found, remove page from cache and make it MRU page of $B_2$.

#### 2.1.4.1 CART

One of the limitations of the ARC and the CAR algorithms is that two consecutive hits are used as a test to promote a page from "recency" (or "short-term utility") to "frequency"(or "long-term utility"), this way not really solving the *Correlated reference problem* 6.

To solve this issue, CART marks its pages with an additional filter bit to indicate whether a page has *long-term utility* (defined as "L") or *short-term utility* ("S"). Pages in $T_2$ or $B_2$ are marked "L", pages in $B_1$ must be marked "S", and pages in $T_1$ can be marked as either. Pages in $T_1$ now can only be replaced if its reference bit is zero **and** its filter bit is "S", if a "L" filter bit is observed, the page is moved to MRU position of $T_2$. On the other hand if a page with its reference bit set is encountered in $T_2$, the page is moved to MRU position of $T_1$. New pages (new to the sets, these can be pages that left all sets, too) are marked "S" always. A transition from "S" to "L" can only be achieved if a page hit in $B_1$ occurs and $|T_1| \geq |B_1|$. This way successive page hits of "S" pages in $T_1$ won't change its filter bit, which can only get upgraded to "L" if the page is demoted from the cache, enters the history set $B_1$, and is referenced again. The rule ensures that two references to the same page are temporally separated by at least the length of the list $T_1$. To simplify things: The list $T_1$ contains pages of type "S" or "L" and approximates "recency". $T_2$ contains pages of type "L" that **may** have "long-term utility".

To complicate things, CART varies the size of $B_1$ (like the target size of $T_1$) in a comparable fashion to quickly adapt to "short-" and "long-term utilization".

CART may sound a little complicated at first sight, but it is actually quite easy to implement and offers the same self-tuning behavior as ARC and CAR do. The authors suggest CART to be more suitable for virtual memory, databases, and file systems, whereas CAR is claimed to more suitable for disk, RAID and storage controllers.

## 2.2 Drops & IDE devices

### 2.2.1 L4IDE

L4IDE [Men03] is a port of the Linux (2.6.6) IDE driver to Drops. It was developed at the University of Dresden.

To be able to adapt a Linux driver to Drops, L4IDE uses and expands a library called DDE (Device Driver Environment [Hel01]), which emulates internal functions and variables of Linux (for example functions that manage processor timings, interrupts, memory areas, PCI / process scheduling and synchronization).

The original IDE core of the driver is left relatively unchanged. The core is responsible for the chip-set management, DMA tuning, the handling of the different device types (e.g. floppy drives, hard disks, CD/DVD drives), and implements the IDE-Interface.

The second part of the original driver, the block driver, is the one where the major changes are made. The block driver is the link between the IDE core and the file-system. It provides waiting queues (per device), handles client requests, and I/O scheduler(s) to optimize disc accesses. The original block driver is mapped to the *generic_blk* interface [Reu01], which was developed as a standard interface for Drops to handle block device drivers in a client/server manner. For data and command transfer IPCs are used exclusively. So L4IDE is essentially a block device server, as intended by the L4 paradigm, which operates in user-mode. Since IPC communication requires a lot of context switches, it is stated that the original Linux driver outperforms L4IDE especially on systems with small L1-caches.

### 2.2.2 BDDF – Block Device Driver Framework

BDDF [Men04] develops L4IDE even further. It tries to generalize the handling of different bus systems, especially SCSI vs. IDE. Both are implemented in Drops and both provide the same user-interface. The actual handling of the particular implementations is distinct, though. The main goal is to provide the user with a unified view to all block devices. For this reason a uniform naming scheme is developed which applies to all sorts of bus systems, devices, and partitions. Furthermore an I/O scheduler interface is provided and a more sophisticated scheduler (elevator) is implemented. This has the advantage that new schedulers don't have to be re-implemented for every bus system. I/O schedulers are allowed to work in parallel.

Secondly, BDDF is concerned with real time disc scheduling and implements a real time model called *Quality Assuring Scheduling* (QAS). For this to work correctly, BDDF offers an own time base to calculate processing times. Since this is not a concern of the swap dataspace manager, please refer to [HLR$^+$01] for a detailed description of QAS.

The IDE core stays the same as in L4IDE, but for the above reasons an own IDE-bus-driver was necessary. The emulation environment is again DDE (2.6). The *generic_blk* is not completely abandoned, but a 100% downward compatibility to the new block driver could not be achieved (for instance flexible block sizes are not allowed in *generic_-blk*, direct device references are also not supported).

Performance issues are the same as for L4IDE. The author states that there is still a lot of optimization necessary to achieve a Linux-like performance, while BDDFs flexibility is claimed to be predominant.

# 3 Design

Currently DROPS offers a physical dataspace manager (DMPhys) only. DMPhys manages the available physical memory of a system, and it implements and extents the L4Env Dataspace Manager Interface. Furthermore page sizes of 4MB (super-pages) are supported. The allocated memory is said to be "pinned", that is, once memory has been mapped to the client, no page-faults will occur while accessing this memory.

The goal of the swap dataspace manager (DMSwap) is to extend the functionality of DMPhys to the possibility to move pages from memory to backup storage (and vice versa) while implementing some sort of replacement policy. The need for physical memory management is still a DMSwap necessity, requiring DMSwap to interact in some fashion with the physical DSM (dataspace manager).

So the question arises how this interaction is to be constructed. One way would be to let DMPhys manage all physical memory, and to increase needed memory dynamically on demand. This requires DMSwap to propagate most of the memory Dataspace Manager Interface calls to DMPhys (especially page faults), also every allocated dataspace would have to be created at DMSwap (for internal page management) **and** at DMPhys (for its internal page management), which seems to be redundant. If the physical memory is exhausted, the replacement policy provided by DMSwap selects pages for replacement. These pages would simply be brought on backup storage and unmapped by DMSwap, thus causing a page-fault on next reference, which gives the swap manager the opportunity to transfer the page back to memory. The advantage of this is that neither the *dm_generic* nor *dm_memory* interfaces would have to be completely re-implemented. Many calls could just be propagated to DMPhys instead, resulting in a smaller (easier to maintain) code-base. On the other hand, this strategy introduces a significant IPC overhead, since any client call (IPC) to DMSwap has to be propagated to DMPhys (another IPC) and all the way back, summing up to at least four IPCs.

A better solution would be to allocate just one (large) dataspace at DMPhys and completely map it to DMSwap. The DSM then uses this dataspace to create new client dataspaces on demand, and maps the appropriate pages to clients as needed. Pages that are selected for replacement are again unmapped at all clients and copied on backup storage to create space for new pages. This way, nothing has to be forwarded to DMPhys once the dataspace has been mapped to DMSwap, and a lot of IPCs are saved. Only one dataspace is created at DMPhys avoiding the overhead of the above solution. The (small) drawback is that the *dm_generic* as well as the *dm_mem* interface have to be completely re-implemented, resulting in some code duplication since many interface functions just stay the same.

Since the second solution results in significant performance and space advantages, it is the one that was implemented in DMSwap. The (large) physically allocated dataspace will from now on be referred to as *swappable dataspace or ds_swap.*

DMPhys provides the possibility to handle different memory pools, which can be used to reserve certain memory areas for special purposes. Swappable memory is such a special purpose. To give the user the opportunity to manage memory areas using different policies, DMSwap operates on just one (user defined) memory pool provided by DMPhys and uses **all** the memory available in the given pool.

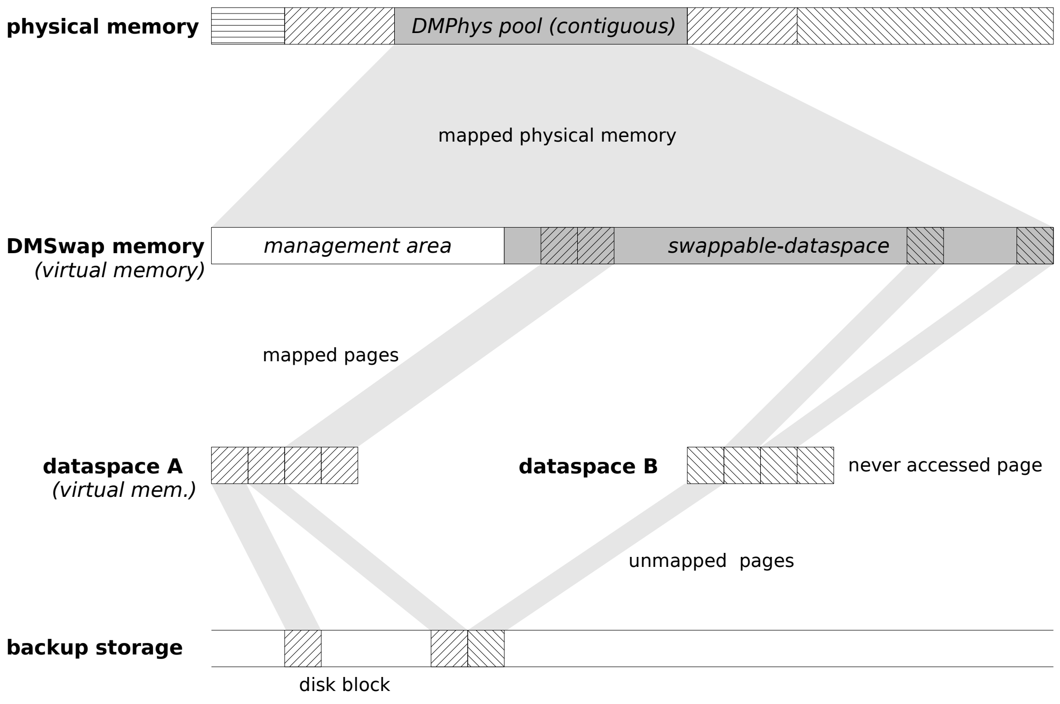

***Figure 3.1:*** Simple (not proportional) model of the memory management structure implemented by DMSwap. The first two pages of dataspace *A* are swapped out, while the remaining pages are still in memory. The last page of dataspace *B* was never accessed.

To to be able to fulfill its goals DMSwap needs to solve two major problems. The first one is the type and support of backup storage devices provided by Drops, the second one concerns the replacement policy[1]to be represented. Both topics will be examined next.

[1] DMSwap aims to define an interface for multiple replacement strategies, for now. This way it tries to stay open for "real life" adoptions as well as optimizations.

## 3.1 Block devices

Two approaches were described in section 2.2. Both L4IDE and BDDF provide sufficient functionality to implement a swap-server. But the choice that has been made is in favor of the BDDF approach. While even the author of BDDF claims that there is still some work necessary to advance BDDF into "real life" behavior, it still seems to be a sophisticated approach.

**Arguments in favor of BDDF**

- BDDF is a generalized approach to the block device handling in DROPS, even SCSI may be supported in the future
- Flexible block sizes are supported by BDDF, giving the opportunity to use the actual physical page size as transfer unit between memory and hard-disk
- BDDF defines a unified naming scheme for all sorts of block devices (instead of device numbers directly adopted from Linux as in L4IDE), which will hopefully be accepted in DROPS
- A more sophisticated I/O scheduler (elevator) is provided, allowing to read/write data more efficiently; priorities are available as well

Most of the above advantages are exploited by DMSwap. Since BDDF right now supports IDE devices only, these are the ones to use. For backup storage a whole hard-disk partition (or a complete raw disk) has to be provided. DMSwap currently allows just one device to be specified. The data is written in raw block units, no file-system layer is implemented. Since BDDF allows to define a block size that can be a multiple of the sector size (usually something like 256 Byte), the block size was chosen to represent the actual system page size (which is 4KB on x86). This way, whole pages can be written to one disk block, so all page data is stored/read sequentially.

### 3.1.1 Block device super-page handling

Super-pages are about 1000 times the size of a regular memory page. DMSwap attempts to provide super-page support. Since these pages require, in the above scenario, exactly 1024 (on x86) disk blocks, the problem of fragmentation arises. The intention of the swap-server is to maximize the speed of hard-disk I/O through sequential storage of super-pages, allowing to read/write the data in an elevator-like fashion (also known as SCAN). For this to work, normal pages are not allowed to "mix up" with super-pages. The idea is simple: Super-pages are written starting from the beginning of the device, while normal pages are written starting from the end of the device. So super-page blocks and normal page blocks grow in the opposite direction. If both areas overlap, the device is exhausted (even though there may be several blocks available in the normal page area). This may waste a little disk space, but on the other hand increases performance.

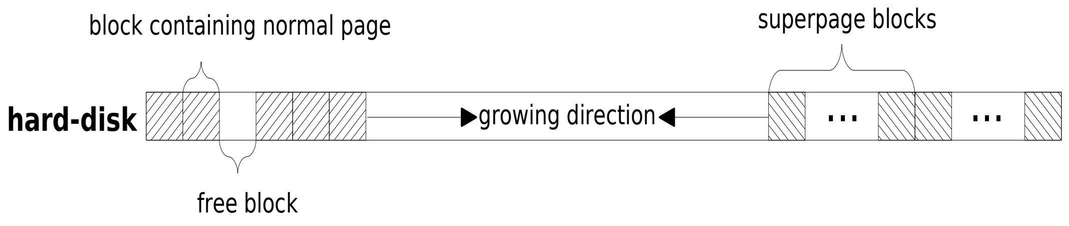

***Figure 3.2:*** Hard-disk layout showing normal page and super-page areas.

## 3.2 The replacement policy

As the attentive reader might have guessed, one of the algorithms proposed in 2.1 has made its way into the swap-server, for all of algorithms described above claim to outperform LRU. All algorithms exploit the "recency" as well as the "frequency" features of a given workload. Differences arise when one asks the self-tuning question.

As described in 2.1.1, 2Q introduces two parameters ($K_{in}$ and $K_{out}$) (*Note:* "Fixing these parameters is potentially a tuning question ..." [JS94]). The values of these parameters have to be chosen *offline* (or *a priori*) In [MM03][2], the performance for different cache sizes and workloads was measured, and it is stated that no choice of $K_{in}$, $K_{out}$ worked uniformly well. The size of the $L_{hirs}$ set suggested by the authors which, is 50% of the cache size for both parameters "is almost always a good choice".

Almost the same arguments can be used for the size of the $L_{hirs}$ set of the LIRS algorithm (2.1.2). The authors suggested size of the $L_{hirs}$ set which is 1%, has been found to work well on workloads that capture "a notion" of frequency, but not for those that are found to be more LRU-friendly. Furthermore LIRS requires a certain "stack pruning" operation, that in the worst case may have to touch very large numbers of pages. So, the claimed constant time overhead is more or less the expected case rather than the worst-case.

ARC (2.1.3) and CAR (2.1.4) have been found to be completely self-tuning, which is achieved by the "online" tuning of the target size of set $T_1$. On a hit in the history set $B_1$ the target size is increased, on a hit in $B_2$ it is decreased. Also ARC is *scan-resistant*, but doesn't really improve on the *Correlated reference problem* 6. This can be achieved with a strategy described in 2.1.4.1, which could also be adapted to ARC. A disadvantage of both algorithms is the requirement to keep track of twice the number of pages the cache is able to keep.

A disadvantage of ARC is that on every page hit the page must be moved to MRU position in some set (1, 2), this is usually a performance killer in virtual memory setups as is dealt with here. CAR now tries to approximate ARC in a clock-like fashion, reducing the time overhead significantly, while not delivering the same good replacement choices as ARC does.

[2] pages 117–118

As the result of the above argumentation CAR would be the algorithm to implement in DMSwap. CAR depends on the ability to read and reset the access bits of page table entries, which hasn't been supported by *Fiasco* at the time DMSwap development began. The behavior could have been emulated, but the cost for that was found to be the same as for implementing ARC. And since ARC is a little better in making replacement decisions, it is the chosen one.

### General policy assumptions

- DMSwap implements ARC as a "global" replacement algorithm, i.e. any page in the swap dataspace can be selected for replacement[3]
- A "demand paging" scenario is assumed, pages are mapped to the client when a request is made (page-fault or explicit mapping call) only
- No "pre-cleaning" is supported where dirty pages, that are likely to be replaced next, are identified and brought on backup storage beforehand
- No "pre-paging" is assumed, i.e. no pages are loaded before a task is started (this behavior is implied by "demand paging")
- The approach described in 2.1.4.1 is not implemented (for now)

## 3.3 How to emulate page reference recognition in Drops

As described above, *Fiasco* didn't support the access to the necessary page table entry bits at the time. Since this feature is actually required by ARC, an emulation had to be proposed to be able to monitor the page accesses.

Since it is way to costly to monitor the references to a page all the time, an approximation is introduced next. The basic concept is to count/monitor page references using page-faults. When DMSwap maps a page to the client, this page usually doesn't produce any page-faults in the future. If the page is unmapped this behavior changes. On next access to the given page, a page-fault occurs that is translated into a dataspace fault, and dataspace faults in Drops are handled by the appropriate dataspace manager, which is in that case DMSwap. This way the swap-server is informed about a page access and is even capable to determine if the causing fault refers to a read or write access of the given page. In the next step the necessary re-ordering of the replacement algorithm sets is performed (if on disk the page is brought back to memory and possible replacements are initiated). Lastly, the page is mapped back to the client.

For the sake of performance the "unmapping" procedure is performed in a cyclic way (right now every second[4]), using a separate thread which unmaps all the pages mapped at all the clients of DMSwap. It is expected that just a considerably small portion of the DMSwap-managed pages are referenced during each "unmapping-interval" resulting in an (hopefully) acceptable performance. The thread is started dynamically when a

[3] as opposed to "local", where only pages of the task/process causing the page-fault can be selected
[4] This value was adopted from the Linux implementation of kswapd.

certain memory usage threshold is exceeded and is stopped when the memory usage drops below another threshold.

## 3.4 Management structures

Taking a closer look, one can find two important management structures necessary to implement a swappable-dataspace manager. Firstly, the replacement algorithm requires extra data to function correctly. This information includes for example, to which dataspace and offset in the dataspace does the monitored page belong to, the location of the page in the *swappable-dataspace*, and in which set (ARC specific) the page is currently located. All this has to be provided for every page overseen by the replacement algorithm. These pages are said to belong to the current *working-set*[5]of the replacement algorithm.

Secondly, the swap-server has to keep track of all the pages in the *swappable-dataspace*, **and** all pages that are currently not in memory (i.e. that reside on backup storage or never-mapped-pages, where memory has been requested, but that memory hasn't been granted because no-one has referenced (demanded) these pages, yet). Required information is again the page identification (owning-dataspace and offset in that dataspace), the possible location on disk, the state of the page, and maybe some reference to its replacement algorithm counterpart (if present). This information will be called *meta-information* in the following.

We define the maximal number of pages the *swappable-dataspace* can hold as $c$ (to stay consistent with ARC) and the maximal number of pages DMSwap can manage as $d$. Following invariant will always hold[6]:

$$d \geq c \tag{3.1}$$

### Working-set handling

The necessary information of a page that resides in the working-set will be called *working-set entry.* ARC requires as many as $2c$ such entries to keep track of the same amount of pages. Since the working-set will be referenced, updated, and reordered very frequently and because of invariant 3.1, it was decided to keep this information in memory. The maximum size needed is calculated by DMSwap at startup and reserved in the *Management Area* (Figure: 3.1) of the given DMPhys pool.

The authors of ARC state (as mentioned before) that the required size doesn't exceed 1% of the total managed memory. For the actual numbers in DMSwap please refer to 5.1.

[5] *Note:* This differs from the definition in literature, where a working-set is defined as the pages that belong to a single process or task, aiming at a "local" scenario.
[6] If the machine is blessed with a hard-disk

### Meta-information – a notion of hierarchy

In contrast to the limited size of the working-set, meta information has to be raised for all the pages handled by DMSwap. This introduces a significant overhead to the system, which the swap-server tries to minimize by making this information swappable as well. The idea is that DMSwap creates a special dataspace to keep the meta information of its client dataspaces. This dataspace is called *meta-dataspace* and is located in the swappable-area. The pages of this special dataspace contain the meta-information of pages from other dataspaces and are treated like any other page in *swappable-dataspace*, meaning they are replaceable and they are overseen by the replacement policy. The meta-dataspace can be said to be an actual client to DMSwap, because it is treated like any other client (with one exception, see below). Since the number of pages a client-dataspace owns can be very large (dataspaces have an address space of 32 bit – in theory), DMSwap must be able to handle a substantial amount of pages per dataspace, while not wasting much memory on small-sized ones. To be able to fulfill this goal DMSwap creates an *index page* for every newly allocated dataspace. This index page is located in the meta-dataspace and contains references to the actual meta-information-holding pages. Meta-information pages will be created as needed. This way a dastaspace, simply needs to reference its index page to be able to retrieve the meta-information for a given page.

The problem that arises for the meta-dataspace is, that its pages also require meta-information (especially if they have to be located on hard-disk). Therefore another dataspace is created that holds the meta-information of the meta-dataspace pages. The difference is that it is located in the Management Area, therefore always resides in memory and is not affected by the replacement policy. This dataspace is called *static meta-dataspace*.

A basic scheme of the meta-information management is shown in Figure 3.3. The boxes represent pages. It can be seen that the first page of the static meta-dataspace keeps information about as many pages as the number of meta-information that fit into this page. The same can be observed for the pages in the meta-dataspace. DMSwap allocates only the number of meta-pages needed by the pages of a dataspace.

## 3.5 Super-page handling or why to break the policy

Super-page handling in DMSwap is of a somewhat basic nature. The large size of these pages (in comparison to the hardware-provided page size) demands a special treatment throughout the swap-server. Even though DMSwap supports these page sizes, the effect on the performance of the system is unknown, and may be rather severe. It is also likely that super-page support will be restricted or may even disappear in the future.

For now super-pages are kept super-page aligned and stored contiguously. The page beginnings are marked $s_x$ in figure 3.4. This is called *super-page-area* in DMSwap terminology. Normal pages can be placed at any position in the swappable-dataspace, super-pages can only be placed in super-page areas. On a new super-page request, DMSwap searches for a free super-page-area. If a free area is found, then the page is placed into that area. On the other hand, if no free area is found, two possibilities arise.

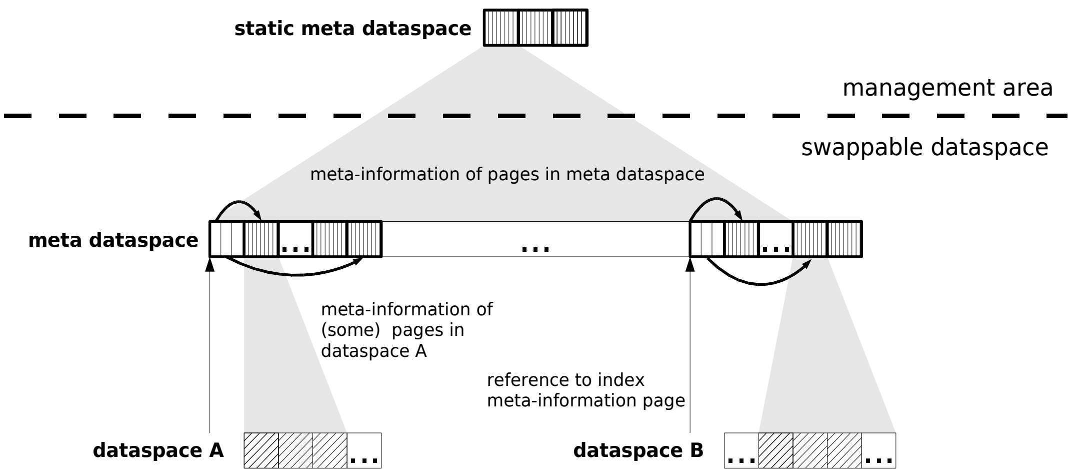

***Figure 3.3:*** Management of meta-information by DMSwap. Dataspaces *A, B* reference their index pages in the meta-dataspace, which hold references to the actual pages containing the needed meta-information.

The first one would be to simply return an "out of memory" error, but on second thought this situation can even occur if the memory hasn't been exhausted yet. So better don't return that (second possibility), since DMSwap is capable to swap out pages.

To support the above postulates a list counting normal pages per super-page-area is kept. In case no free area is found, DMSwap reviews the replacement algorithm to check if the next page to replace is a super-page. On success it replaces the page found. On failure all areas that contain normal pages are searched, the (currently implemented) algorithm then selects the super-page-area containing the minimum number number of normal pages. All pages in this area are then scheduled for swapping. (One could also imagine different technics. For example, the algorithm could select the super-page-area that has the highest average age.) This behavior overrules and compromises the replacement policy and leads to a situation where essentially "hot" pages, which have been brought to backup-storage right-afterwards may to have be read-in. In the worst case a complete super-page-area containing normal pages has to be swapped out. If everything fails the replacement algorithm is forced to emit a super-page of its choice. Usually that should be the super-page that would selected by the algorithm next, but the choice is left entirely to the replacement policy.

The replacement of super-pages and super-page-areas differs from normal page replacement. DMSwap always reserves one super-page area. On page replacement (super-page or super-page area) the newly requested super-page occupies this area. In the meantime, the swap out procedure of the area to be replaced is initiated. If this process is finished, the swapped-out area becomes the new reserved area of the swappable-dataspace. Figure 3.4 demonstrates the this behavior.

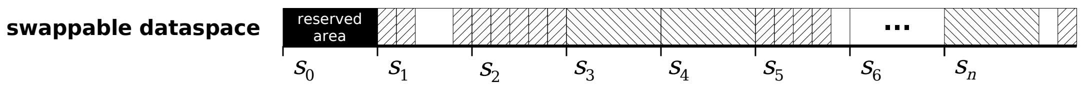

***Figure 3.4:*** Example memory usage pattern in the swappable-dataspace. Straight rectangles mark normal pages while wide blocks symbolize super-pages. $s_x$ marks the beginning of a super-page aligned areas.

Imagine there is a super-page request in figure 3.4 and the replacement policy preview doesn't return a super-page to be swapped out next. In this case the search for a super-page area containing the minimal number of pages is initiated. The returned area will be area $s_1$. So the new super-page will be allocated in area $s_0$ (reserved area) and $s_1$ will be the new reserved area after write-out is finished.

# 4 Implementation

This chapter describes some details of the design implementation. In general DMSwap aims to minimize the overhead that is necessary for page- as well as hard-disk management by trying to keep things as simple as possible. Also DMSwap completely supports the *dm_generic* and *dm_mem* interfaces, with the exception of the `l4dm_mem_ds_is_contiguous` call, because it is very unlikely that a dataspace will be allocated contiguously by DMSwap (for this is the very nature of a global replacement policy). This is why the stated call always returns false (0).

## 4.1 DMSwap internal threads and requirements

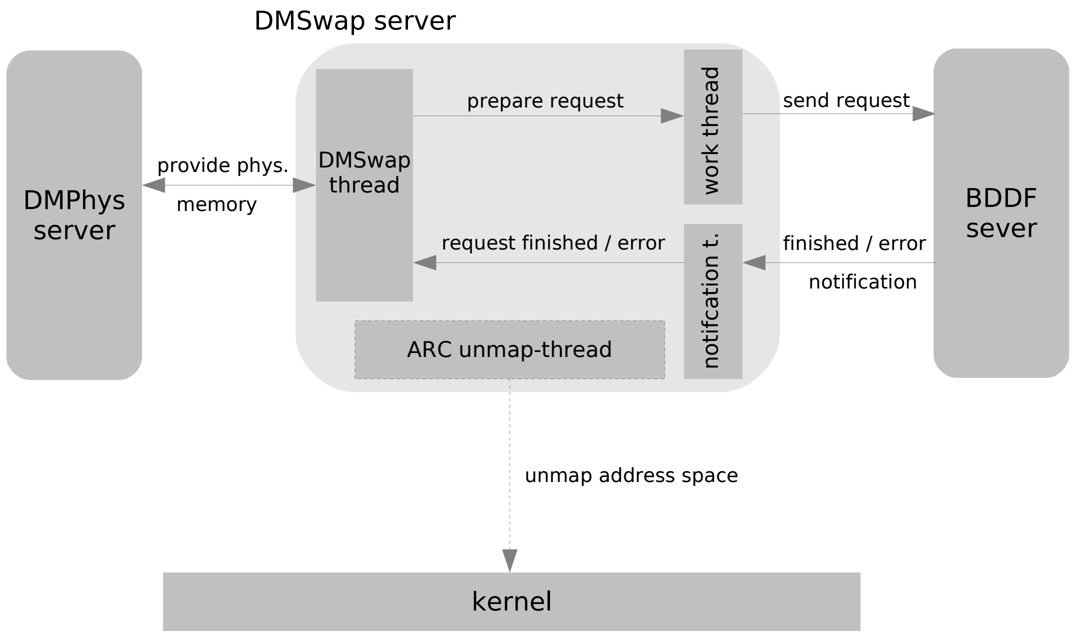

***Figure 4.1:*** DMSwap threads showing interaction with its environment

Besides the DROPS standard threads that are started by every user-mode server (l4rm, semaphore thread) and the DMSwap-thread itself, BDDF requires two client threads to be started by the swap-server. The first one is called *work thread* and is responsible for sending requests to the BDDF-server, the second one is the *notification-thread* which receives notifications about finished requests from the BDDF-server. Also ARC requires a specific *unmap-thread* (as described in section 3.3), to emulate page reference counting it periodically executes the "unmap" system call. This thread is started and stopped as needed.

DMSwap depends upon DMPhys for physical memory allocation and BDDF for hard-disk support. To function correctly BDDF requires *l4io*. This is an implementation of of the *generic_io* interface that defines I/O access in DROPS.

## 4.2 Meta-information

As described in section 3.4 the swap-server keeps meta-information on a per-page-basis. This information has to be kept for pages in memory as well as for pages on hard-disk. But the information needed is rather limited. Here is the actual data structure:

```
typedef struct dmswap_meta
{
    l4_offs_t offset;
    dmswap_wkset_entry_t * w_entry;   /* pointer to work-set or NULL */
    l4_uint32_t block_nr;
} dmswap_meta_t;
```

The `offset` is the one in the owning client-dataspace. The `w_entry` points to the entry in the work-set of the replacement algorithm, if the page is overseen by the policy. It is used to retrieve the information at which position the page resides in the swappable-dataspace, for "locking" and "unlocking" of pages (see below), as well as for the necessary re-ordering of the sets ($T_1, T_2, B_1, B_2$) of ARC, on page references.

The `block_nr` (block number) serves a double purpose. The three most significant bits encode the status information of the watched page, while the least 29 significant bits hold the actual block number on hard-disk. To set/access the block number of a meta structure the macros `SW_SET_BLOCK` and `SW_GET_BLOCK` are available. If a page already owns a disk-block `SW_HAS_BLOCK` returns one. Since the chosen block size of the hard-disk/partition is equal to the page size (section 3.1), only one block number per page is needed. This holds true even for super-pages, because they are written sequentially to disk (section 3.1.1), and only the first block of a super-page is required to retrieve the whole page from disk. Twenty-nine bits for block numbers are also sufficient, because assuming a page size of 4KB the supported hard-disk size is still about 2048 Terra-byte.

**Page states:**

- SW_NEW: New page; the memory for the page has been granted, but t he page hasn't been allocated by DMSwap yet, because no reference to that page occurred
- SW_PRESENT: The page is in memory and hasn't been written yet (only reads occurred; this state is also known as "clean" state)

- SW_DIRTY: The page is in memory and it has been written on
- SW_LOCKED: The page is in memory and "locked" (i.e. cannot be replaced)
- SW_SWAPPED: The page resides in backup storage and not in memory[1]

The above strategy gives DMSwap the opportunity to simply delete clean pages, if they reside on hard-disk (block number greater than zero and state SW_PRESENT) on page replacement, resulting in some gain in performance.

### Meta-information indexing

What about finding the meta-information for a given dataspace and offset? Answer: "Easy as pie." Or cake if you prefer. Firstly the given offset is rounded to a multiple of the page size (either page or super-page size depending on the dataspace). As shown in figure 3.3, every dataspace owns an index page in the meta-dataspace, each index points to some meta-information page which is again located in the meta-dataspace. Since the number of meta-information structures that one meta-information page can hold is known in advance, the appropriate index that points to the right meta-information page can be determined. For the very same reasons the location of the meta-information structure in the indexed page can be determined as well. This introduces a possibly constant time overhead, with the exception that when the needed meta-page resides on hard-disk, it has to be swapped-in first.

### Notes on the meta-information dataspace and its static counterpart

As explained earlier, the pages of the meta-dataspace are located in the swappable-area, while the pages of the static meta-dataspace are kept in the management area. To get an impression of the memory consumption of this approach, again imagine a L4 page size of 4KB. In this case one page can hold 341 ($4096Byte/12Byte$) meta-information structures, and since the static dataspace tracks the page information of the meta-dataspace, $341^2$ (over 100.000) pages can be allocated before one 4KB page in the static dataspace is filled.

The meta-dataspace also owns a free-list, to be able to reclaim freed memory in itself. The list is a classical bitmap (as all the free lists in DMSwap), thus requiring one bit of information per page of the dataspace. From the position in the bitmap the offset in the meta-dataspace can be calculated directly. The maximum size of the static dataspace is computed at the startup of the swap-server, a free-list is not necessary because the position of a meta-page in the static dataspace can be determined from the offset in the meta-dataspace.

[1] A page can be on hard-disk and in memory at the same time.

## 4.3 Replacement algorithms

Usually a replacement policy keeps track of one or more (in case of ARC four) sets of pages. As described in section 3.4, information per overseen page is needed. The actual data structure is shown below:

```
typedef struct dmswap_wkset_entry
{
    l4_uint32_t ds_id;
    l4_offs_t offset;
    l4_offs_t dmswap_offset;
    l4_uint32_t where; //ARC specific (cache id)
    struct dmswap_wkset_entry* next;
    struct dmswap_wkset_entry* prev;
} dmswap_wkset_entry_t;
```

As can be seen, the structure is organized in a simple double-linked list, which enables the policy to reorder the appropriate sets[2]. The `where` field is used to mark the set-affiliation. `ds_id` and `offset` again identify the watched page, while `dmswap_offset` describes the actual location in the swappable-dataspace.

The above structure can be used to implement a variety of replacement algorithms (2Q should be as possible as LRU). For debugging purposes, FIFO has been implemented besides ARC.

### ARC implementation and the algorithm interface

The interface a replacement policy has to implement is briefly and exemplary for ARC described next:

| | |
|---|---|
| `wkset_init` | Initialization |
| `wkset_new_entry` | Allocate and return a new page entry |
| `wkset_set_ref` | A page entry has been referenced |
| `wkset_del` | Delete a page entry |
| `wkset_lock` | Lock a page, so it can't be replaced |
| `wkset_unlock` | Unlock a page |
| `wkset_check_super` | Check if a super-page will be replaced next |

The `wkset_init` function is called during DMSwaps-startup, providing the maximum number of pages ARC has to take care of. This allows ARC to determine its cache size, which is twice the number of pages ($2c$) the swappable-dataspace can hold. If a new page is allocated by DMSwap `wkset_new_entry` is called and a new entry is returned. Internally ARC keeps a slab-cache for all page entries.

`wkset_set_ref` is called when DMSwap encounters a page reference in the way described in section 3.3. This is the heart of ARC and contains a straight forward

[2] Most algorithms just move entries to MRU or LRU positions.

implementation of the algorithm described in [MM03] (page 123). To identify in which set a referenced page resides ,the `where` field of the page entry structure is used (values can be: ARC_T1, ARC_T2, ARC_B1, ARC_B2 or SW_WKS_NEW for an unknown page). After this identification the page is reordered/inserted according to the rules of ARC. If a page has to be replaced, its page entry is returned, allowing DMSwap to take care of the further replacement procedure. Also ARC checks if the unmap-thread has to be started or stopped (section 3.3).

Super-pages are treated like any other page, so a super-page occupies one page entry as well, it just uses up a bigger portion of the cache allowing ARC to perform replacement conveniently.

`wkset_del` is called when a client-dataspace (or a part of it) is released, ARC then frees its page information structures.

Locking and unlocking is implemented by just removing the given page from the set it resided in ($T_1$, $T_2$ only). The page entry is not released, and its `where` field is set to locked (SW_WKS_LOCKED). Also the cache usage is **not** decremented. This way the page cannot be replaced by the policy. The unlock operation shows the opposite behavior, the page is moved to the MRU position of the set ($T_1$ or $T_2$) it originally belonged to.

`wkset_check_super` is used by DMSwap to test if the next page to replace will be a super-page (see section 3.5). It returns true if that is the case, false otherwise.

## 4.4 Hard-disk management

The sections 3.1 and 3.5 define three cases of data transfer between memory and hard-disk. They are: reading/writing of pages from/to disk, reading/writing of a super-pages from/to disk, and the writing of (super-)page areas to disk. Every single one of these cases has been implemented separately to optimize handling and performance. In general all write operations to disk are performed asynchronous, while reads are synchronous. The latter even counts for super-pages, which may be changed in the near future. As mentioned before, DMSwap stores only one block number per page since the block size was chosen equal to the hardware page size (and super-pages are written sequentially to disk).

### Pages

To handle normal page-requests DMSwap keeps a waiting queue. On replacement the chosen page is copied into this buffer, a free disk block is searched, and the write-out request is initiated. The replacing page can populate its memory right after the request was scheduled. Because of this producer/consumer behavior the synchronization is implemented by semaphores.

### Super-pages

The write-out of super-pages differs significantly, because every sort of buffer would take up entirely too much space. A super-page is written directly out of its location in the swappable-dataspace. The replacing page occupies the "reserved area" (3.5) in the meantime. After the write operation has been finished, the area of the replaced page then becomes the new "reserved area". Also DMSwap has to send several requests to the BDDF-server, because there is a certain limit of sectors per request the driver can handle. This limit is not defined by BDDF, but by the original Linux driver. In the current version of BDDF, this limit is 256 sectors per request (for most IDE devices), and the most common sector size is 512 byte. This way a (4MB) super-page writing has to be divided into 32 requests.

### Super-page areas

To be able to locate all pages in a super-page area, another lookup facility is necessary. For this reason DMSwap keeps an array of page beginnings (`reverse_lookup`), resolving these beginnings to `ds_id` and `offset`. If DMSwap decides to backup a super-page area (area containing the least number of pages (3.5)), a search for all (valid) pages in the given area is initiated. In the worst case the number of pages encountered will be *super-page size / page size* (on x86 over a thousand). The pages thus found are then scheduled for replacement. Meta-information pages that contain meta-information of other pages to be swapped-out may also reside in the selected super-area. As a consequence the write-out process has been divided into two steps. In the first step, normal pages are handled. For each page a disk-block is located, its state is updated (i.e. its meta-information), and the write-request is send. In the second step the same is performed for meta-information pages. This is necessary because the meta-information of normal pages could reside in the selected super-area as well, and its write-request could have been started already, leading to a possible loss of information. On the other hand, meta-pages of the meta-dataspace are kept in the static-dataspace, which stays in memory permanently.

The disk-requests of meta-information pages are given a higher priority than those of normal pages. So meta-requests can outdistance normal requests. The intention behind this behavior is, that meta-information is always required before a page can be handled, and thus should be placed on hard-disk as quickly as possible.

### Hard-disk block management

Free and used disk-blocks are managed using bitmaps. Each block can be in one of four states (see below), thus requiring two bits of information. Additionally one bit that marks the block, if it is currently involved in a write request, had to be introduced. Initially, there was another page state describing that a page is currently being swapped-out. On completion of the request the state was changed to either present or swapped. This complicated the synchronization between the swap-server and the asynchronous BDDF requests dramatically, mostly because of page state checks. It is possible that the meta-information of the page is located on disk, causing a swap-in and some more state transitions. Also the locking of pages required additional synchronization that even affected the replacement algorithm level. To solve this, the additional bit was introduced. A page finding its state set to "swapped" now checks this bit (indexing the bitmap with its block number), to be able to determine if the write-operation is already finished. If this is not the case, DMSwap just “busy-waits”, without having to test the pages meta-information (which may just find its way to disk) state all the time.

Two important goals are: Don’t swap clean pages that are already on disk; and allow the overwrite of blocks whose pages have been read back to memory, if the disk space turns low.

The possible four block-states defined by DMSwap are:

**Free ($F$):** The block is free.

**Exclusively owned ($E$)** : The block contains a page, which is located on hard-disk and is not in memory.

**Read-in ($R$):** The block contains a page, which has been brought back to memory.

**Overwritten ($O$):** The block has been overwritten by another page.

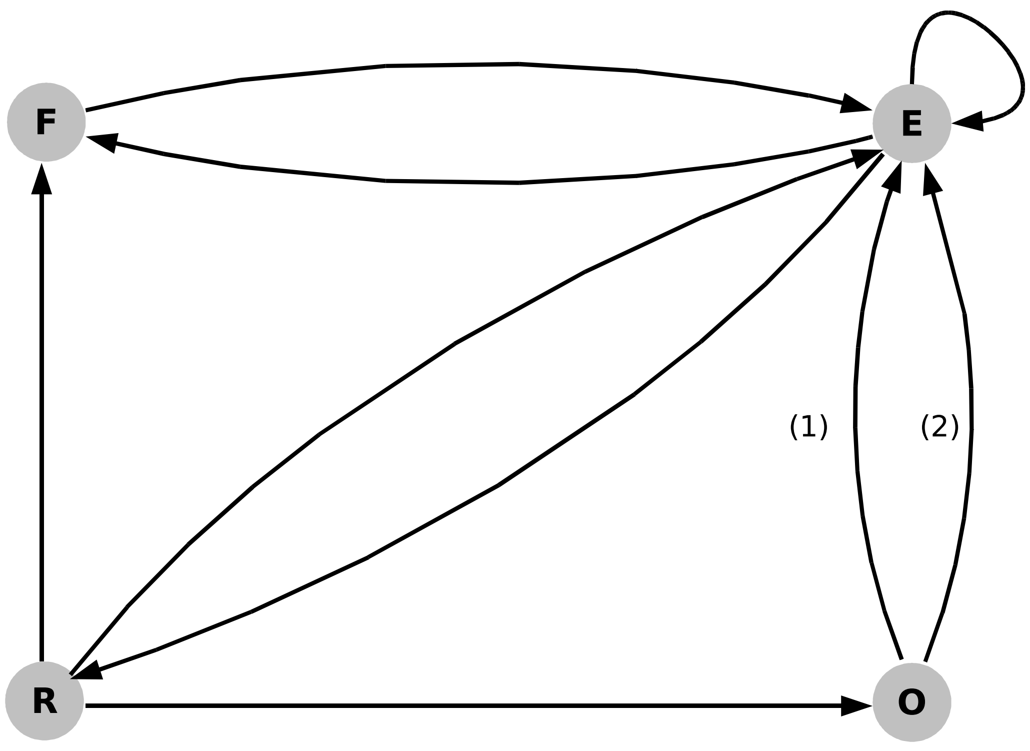

***Figure 4.2:*** Hard-disk block state transition diagram.

State transition descriptions:

$F \rightarrow E$ : The block is free and set to used

$E \rightarrow R$ : The page is read back to memory

$E \rightarrow F$

$R \rightarrow F$ : The page is released

$R \rightarrow E$ : The page is swapped-out again; the block is still valid; no write operation necessary (if the page is still clean)

$R \rightarrow O$ : The original page is in memory and can be safely overwritten by another page

$O \rightarrow E(1)$ : The original page finds its block overwritten, grants the block to the overwriting-page, and begins to search for another block

$O \rightarrow E(2)$ : The overwriting page is read back to memory, and leaves the block ownership to the original page

$E \rightarrow E$ : If a page in memory encounters its block state to be $E$, the block has been overwritten in the meantime, but the page still owns the block; in consequence the page has to be written again

During block search DMSwap always tries to find free blocks first, while still keeping track of $R$ blocks. The $R$ blocks are the fall-back blocks, if no free ones are available. The disk is exhausted when neither a $F$ nor $R$ block can be located.

## 4.5 DMSwap usage notes

There a two ways to use DMSwap. When the server is up, every application can take advantage of the *dm_generic* and *dm_mem* interface calls, with the exception of those calls that determine the dataspace manager to use with the `l4_env_get_default_dsm` call. This call will most probably return DMPhys, causing the request to be handled by DMPhys, not DMSwap. This concerns most of the allocation functions of *dm_mem.* To actually allocate memory, the client should then use the “open” interface-call directly (*note* that the returned dataspace has to be attached to a region by hand).

The other way would be, to use the *l4loader* to start ones application/server. The loader gives an application the opportunity to specify its default dataspace manager (e.g. `ds_manager 'dm_swap'`), thus enabling the application to completely use the generic and memory interface capabilities.

# 5 Performance evaluation

Realistic data cannot be presented here, for Drops is lacking some sort of benchmark to compare results. This is the author's first priority to improve upon the achieved in the future, and this is why this chapter covers some memory and performance concerns, mostly of theoretical nature.

## 5.1 Memory overhead

To illustrate the memory overhead produced by DMSwap, a system with 4KB page size, 512 MB RAM, and 1024 MB hard-disk space is assumed.

### Per page overhead

Each page owns a meta-information structure, a possible entry in the ARC work-set, and an entry in the reverse lookup table. If we assume, that the page, as well as the page that holds the meta-information, reside in memory, then the total overhead is the sum of the meta-structure (12 Byte), the work-set entry structure (24 Byte), and the reverse lookup entry (8 Byte), totaling to 44 Byte. $\frac{44Byte}{4096Byte}$ results in an overhead of about 1.1%, which is almost equal to the percentage proposed in [MM03].

### Dataspaces

Every dataspace allocates an index page in the meta-information dataspace, and at least one page in the same dataspace that contains the actual meta-information structures (figure 3.3).

In the assumed scenario, one meta-page can hold 341 meta-information structures (section 4.2). Each index page is capable to maintain 1024 meta-page references. Thus, the maximum size of a dataspace supported by DMSwap totals to $1024 * 341 * 4KB \approx 1.3GB$, allocating 4KB pages only.

While this management strategy is sufficient for large dataspaces, smaller ones still require an overhead of at least 8 KB, even if the client-dataspace only consists of one page. This is a problem that definitely requires some improvement in the future. The easiest solution would be to omit the index page for non-super-page dataspace sizes smaller than $341 * 4KB \approx 1.3MB$.

### Hard-disk

As mentioned in section 4.4, DMSwap allocates 3 bits per block. The block size in this example will be 4 KB (block size = page size). For the given hard-disk size (1024 MB), DMSwap manages 266144 disk blocks and requires an overhead of $(266144 * 3bit)$ 96 KB, which will be kept in memory. Furthermore, the size of the waiting queue that handles normal page requests is currently set to 128KB (32 pages in this case).

## 5.2 Unmap performance

The unmap-operation is a ARC specific feature. It is used to emulate page-reference-counting in DMSwap. The feature is implemented, using a separate thread, executing the `l4_unmap_page` system call periodically. For every bit set in the size variable of the swappable-dataspace (which is a multiple of the systems page size), a flexpage is build and handed to the `l4_unmap_page` call[1]. Also, the `L4_FP_OTHER_PAGES` map_mask is enabled, causing the unmap-operation to be performed in all address-spaces into which DMSwap-pages have been mapped directly or indirectly.

The `unmap` system-calls are rather expensive, because they iterate through the whole swappable-dataspace address-space[2]. and unmap found pages (and possibly its subtrees) individually.

The fact that after the address-space has been unmapped, all page-references will cause page-faults, effects the performance as well (even though it is the author's hope that only a considerably small fraction of the DMSwap-managed pages will be referenced during each unmap-interval).

*Fiasco* now supports the read of the page-tables access bits. This can be used to implement a CAR-like algorithm, reading access bits at page replacement time. However, the number of pages to investigate can become very large as well, thus producing a lot of IPC-system-call delay. On the other hand, the number of page faults is reduced dramatically, making this approach favorable.

## 5.3 BDDF performance

Performance evaluation, measurements, and a comparison to the Linux IDE-driver are dealt with in [Men04] (chapter 5, pages 60ff). The thesis claims the performance of BDDF to be somewhat worse than the one of the Linux driver, mainly because of the extensive usage of IPCs, and thus necessary context switches in L4.

Additionally, the experience of DMSwap suggests the use of sequential memory areas for larger writes, because BDDF requires a physical address. This address has to be retrieved from DMPhys, requiring even more IPCs.

The maximum number of BDDF-requests DMSwap demands at a time is currently limited to 1024 requests (section 4.4). For this to work the BDDF heap size had to be set to 192 KB[3].

[1] If the swappable-dataspace size is for example 100 MB (0x6400000), than l4_unmap_page will be called three times, with flexpage ($\log_2$) sizes of 26, 25, and 21.

[2] *note:* The address-space of DMSwap is usally larger than the one of the swappable-dataspace

[3] The actual memory requirement per request has not been determined, yet

In general, both BDDF and DMSwap should be marked unstable, for both systems still contain some bugs, and are certainly in need of real-life usage patterns to optimize their behavior.

## 5.4 Measurements

DMSwap has been tested stressing my good old Pentium IV Williamette@1800MHz (8 KB L1-cache, 256 KB L2-cache, 512 MB DDR@266 MHz RAM). The hard-disk used was a 40GB Maxtor (D740X, 1.8 MB cache, 7200 rpm@UDMA 100). For reasons of predictability, the swappable-dataspace size has been set to ≈12 MB (3024 pages). All data was gathered in the given environment.

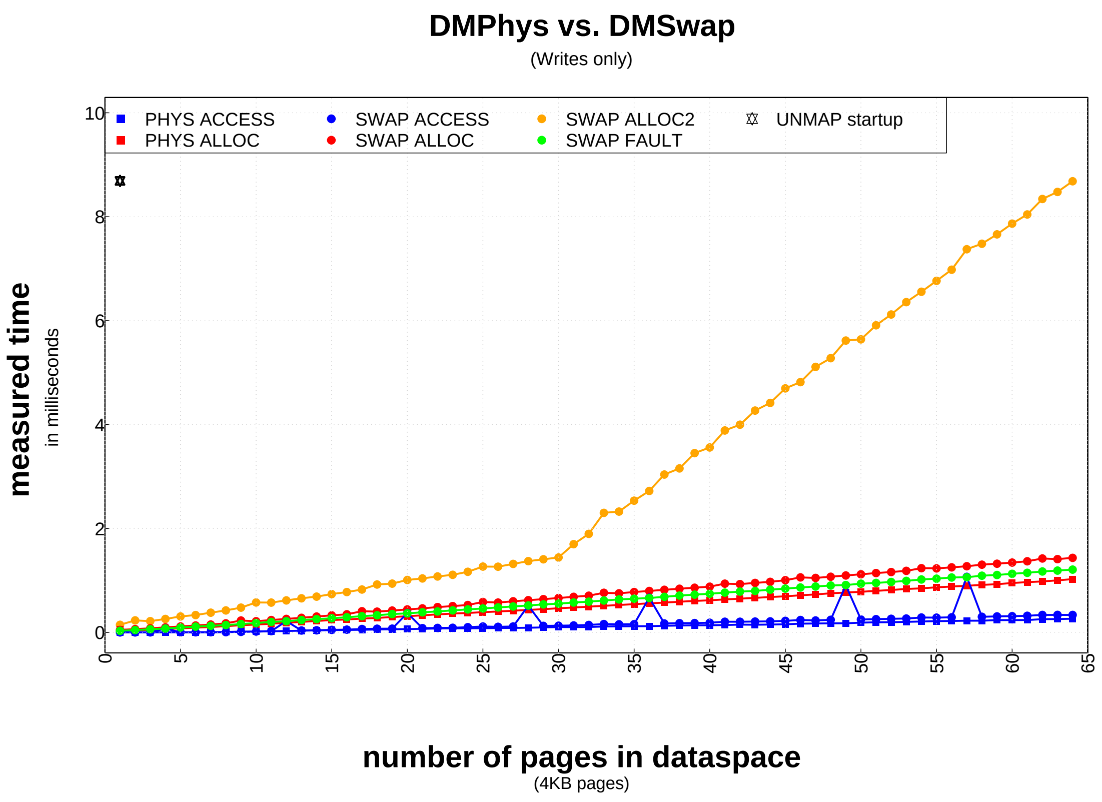

***Figure 5.1:*** DMSwap performance compared to DMPhys

In the measured scenario, dataspaces of sizes 4 KB through 256 KB have been allocated and used. *ALLOC* describes the time it takes to freshly allocate a dataspace of the given size, while *ACCESS* displays the time needed to write data to the whole dataspace. *FAULT* is the time that is consumed by the re-mapping of pages that have been unmapped by the ARC-unmap-thread. The (few) fluctuations in *SWAP ACCESS* are caused by the unmap-thread, which interfered unpredictably.

*ALLOC2* shows the allocation of memory in the case of complete physical memory exhaustion. So, before $x$ amount of pages can be allocated $x$ amount of pages have to be scheduled for replacement. It can be observed, that the ascent of this curve increases after the size of the waiting buffer queue has been exceeded (30 pages). Also notice the time consumed by the ARC-unmap-thread startup, displayed at the first abscissa value.

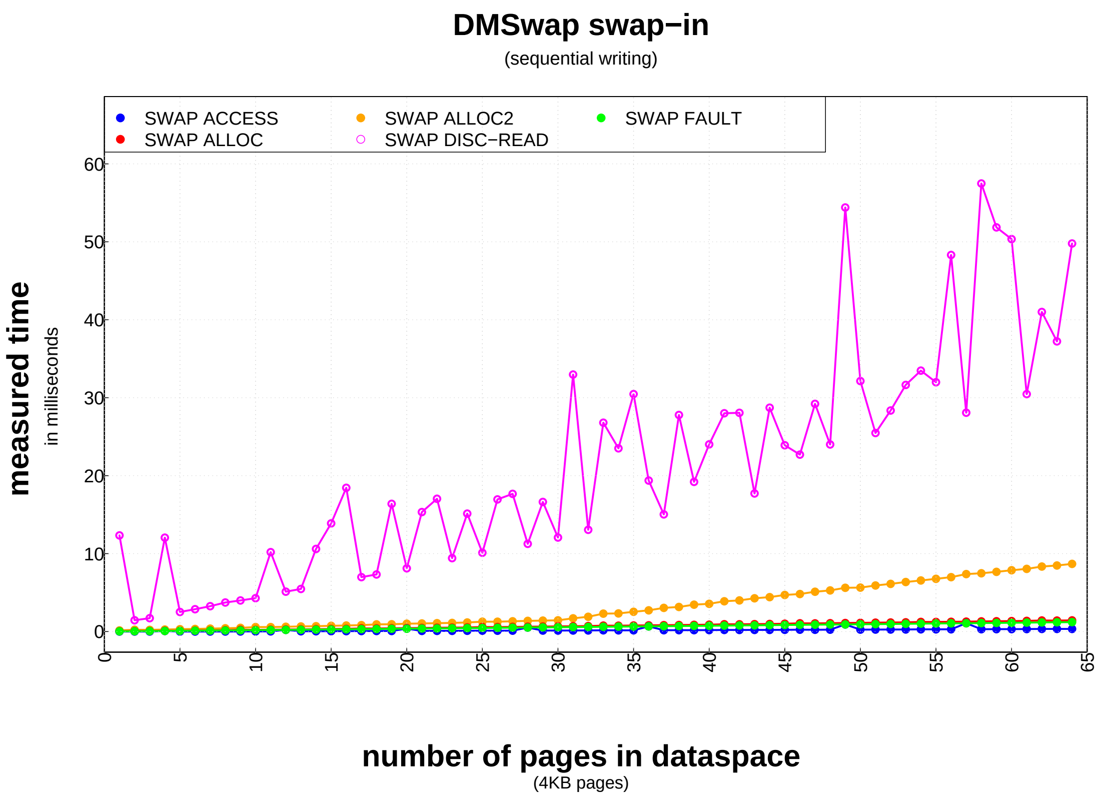

***Figure 5.2:*** DMSwap performance compared to DMPhys

Figure 5.2 shows the same situation as before adding the times measured for dataspace reads from hard-disk; this requires disk-read and write operations. The swaying of *SWAP DISC-READ* is most likely caused by the “elevator” scheduler of BDDF, which is trying to write consecutive disk-blocks first. A mix up between read and write requests resulting in disk-head seeks (since read requests are prioritized), can be a cause as well.

The next plot (5.3) is a real overload situation (remember the setup). Dataspaces of sizes between 5 and 40 MB are requested. The allocation time is measured first, resulting in the displacement of the same amount of memory. Secondly, the read-only performance of pages that reside on hark-disk is given, and finally the same is done with the difference that the pages are dirtied.

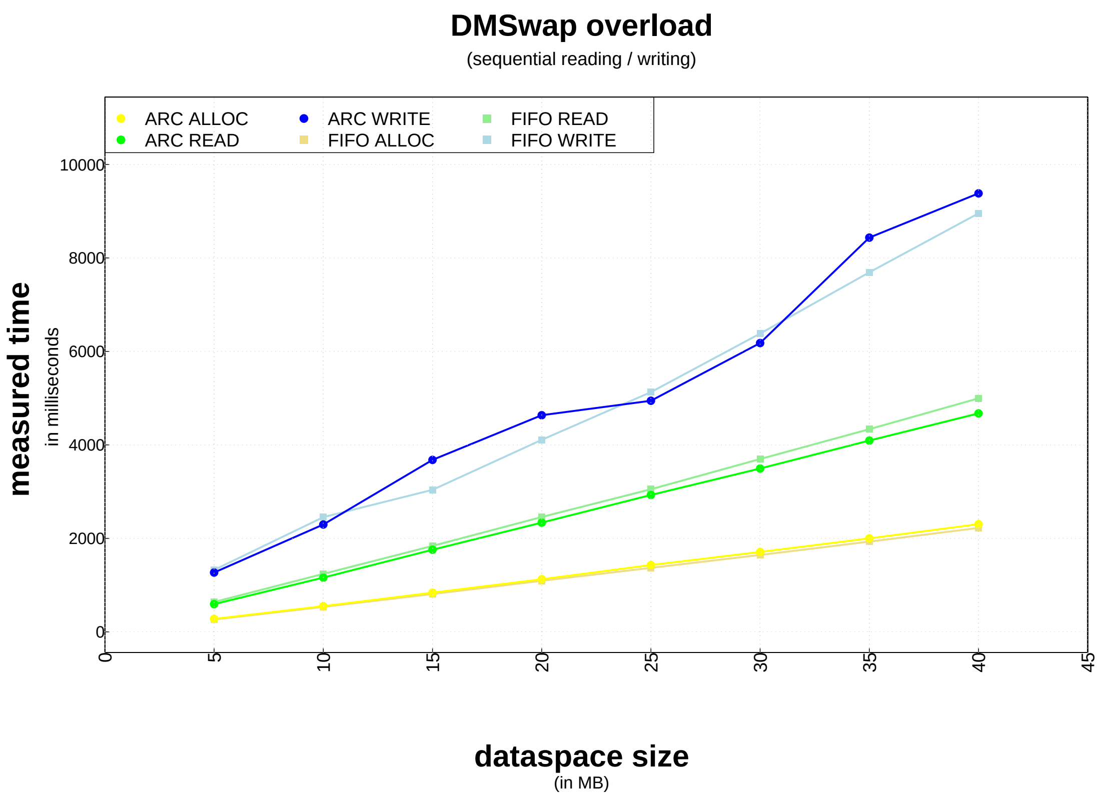

***Figure 5.3:*** DMSwap overload performance

As can be seen, the memory allocation performs best. This is because the swap-out operation is done asynchronously and BDDFs “elevator” scheduler tries to write blocks sequentially. The read-only performance turns out to be better than the write performance, because the replaced pages are “clean” and thus don’t have to be written back to hard-disk. Since the dirtying of pages (*WRITE*) results in a reading in addition to a writing of a hard-disk block on replacement, its last place is obvious. The strange behavior of ARC is unidentified. The performance of FIFO and ARC is, due to the fact that all datapaces are accessed in sequential order, almost the same.

Unmap intervals seem to have a crucial affect on the performance of ARC. Since unmapping (i.e. calling `l4_fpage_unmap`) and the resulting page-fault(s) suffer from some performance losses. The interval of the unmapping procedure turns out to be a “tunable” parameter, which depends on the size of the swappable-dataspace.

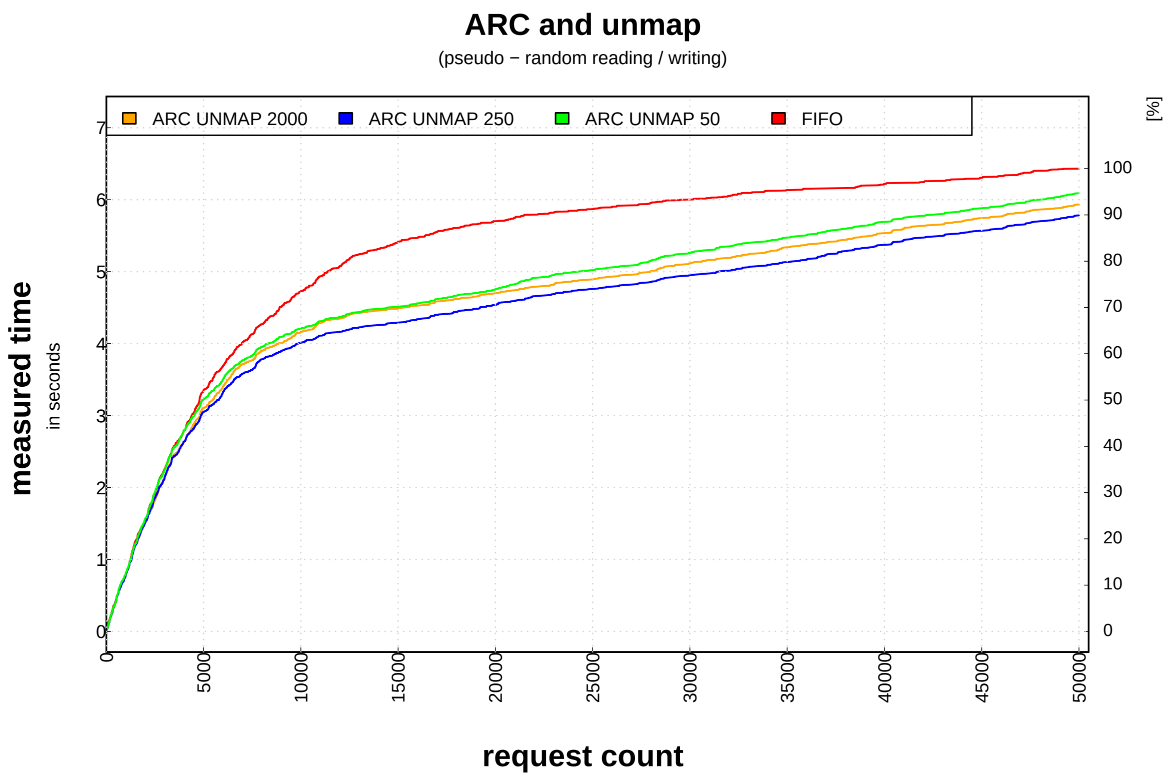

***Figure 5.4:*** ARC-unmap thread influences

Figure 5.4 was generated using three dataspaces, which require twice the amount of the available memory. The first dataspace had a hit probability of $\frac{1}{2}$, the second of $\frac{3}{8}$, and the third of $\frac{1}{8}$. Fifty thousand reads and writes where varied in a pseudo-random (same for all measurements) fashion.

A to short unmap interval (50 ms) results in a worse performance, due too the unmap/map overhead, while a too large interval (2000 ms) leads to a poor adaption of the ARC algorithm, because page-reference counting is inaccurate. It is expected that the measurements shown will become more significant using larger swappable-memory sizes.

The final plot (5.5) demonstrates the superiority of ARC in the case of adaption. The setup is the same as before. Only the dataspace hit probabilities where changed at the 25000th page access. ARC "adapts" to the workload, resulting in a higher average cache-hit-rate.

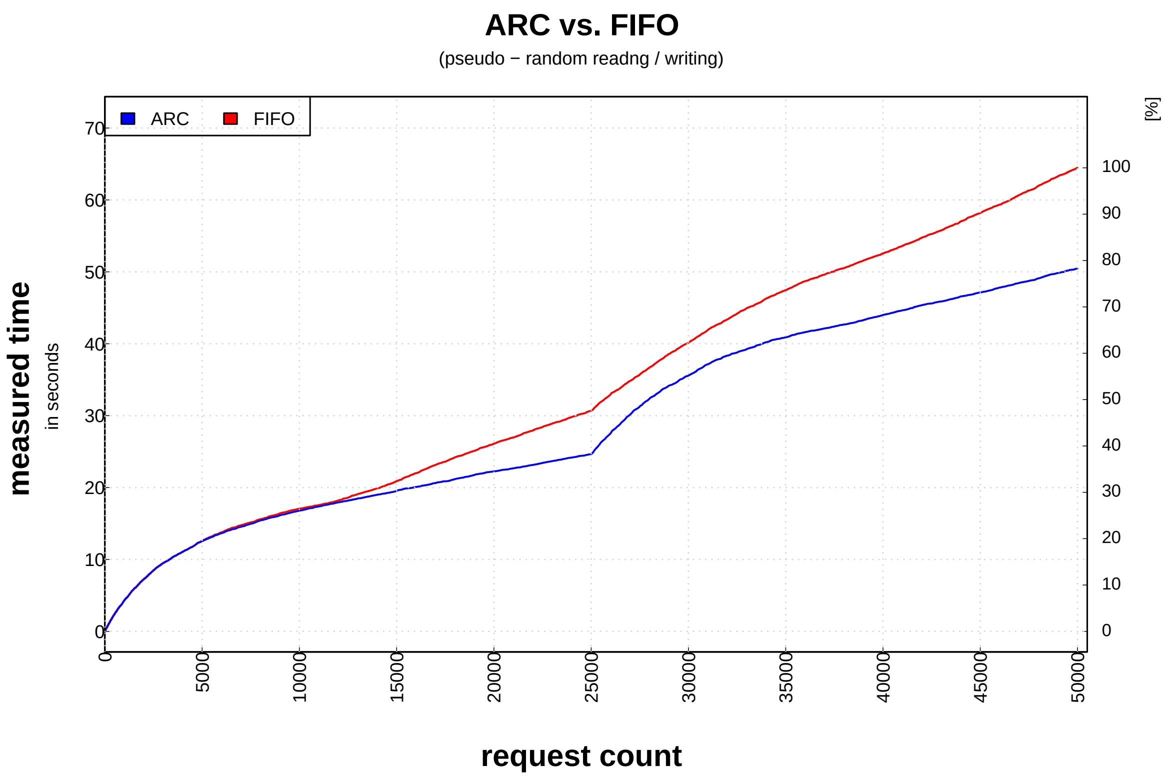

***Figure 5.5:*** ARC adaption

# 6 Summary and outlook

DMSwap has to be seen as the first attempt to provide a page-replacement policy for Drops. The nature of the dataspace manager is more like that of a reference implementation, describing requirements, problems, and solutions developing for Drops. Especially the super-page handling is a topic that requires further development, while also introducing the need for a performance measurement system to be able to improve the swap-server. This should be a top priority issue, which would allow the performance-comparison of different replacement strategies. So the first future step would be to port or reinvent some replacement benchmark for L4.

As can be seen throughout this paper, special super-page-treatment is required. Because of the size of these pages, the handling is very difficult, especially in the case of swapping. One way to improve upon the current behavior, is to treat super-pages like normal pages and scatter them internally. Although this could increase performance, it compromises the concept of super-pages.

It has been outlined that *Fiasco* now supports page-table access bit reads/resets, thus allowing DMSwap to perform better page-reference counting. This also introduces the possibility to implement the CAR (or even the CART) algorithm described in section 2.1.4, providing better performance for virtual memory setups by not counting all page-references at all times. The incorporation of the CAR algorithm into DMSwap-server is also possible at the current state. In general DMSwap should, after some experiences have been made, decide on one replacement-algorithm to support, to be able to optimize the swap-server to the needs of the chosen policy. The author actually doesn't see the need to offer more than one replacement algorithm. This support was mainly implemented to be able to compare different algorithms; once a suitable one has been found, the demand for multiple algorithm support will probably cease.

Another improvement could be to always reserve a certain number of free pages. This way, especially small requests can be satisfied immediately, while replacement or cleaning of pages is initiated later. Of course, one has to determine/measure a adequate amount of pages to reserve. Pre-cleaning would be a "nice-to-have" feature as well. This strategy tries to identify dirty pages that are likely to be replaced next and starts to bring these pages on backup-storage, even before the actual/possible replacement, hopefully saving some write operations on later replacements. Because this approach can waste I/O bandwidth, in the case where the chosen pages become re-dirtied before replacement, the actual performance gains/losses are of some interest.

To sum things up, DMSwap provides a basis for future developments and improvements of the virtual memory management in Drops.

# Glossary

**ARC** Adaptive-Replacement Cache

**BDDF** Block Device Driver Framework

**CAR** Clock with Adaptive Replacement

**DDE** Device Driver Environment

**DMA** Direct Memory Access

**DROPS** Dresden Real-time OPeration System

**DSM** Dataspace Manager

**IPC** Inter-Process Communication

**IRQ** Interrupt Request

**IRR** Inter-reference Recency (used in LIRS)

**L4** A mikrokernel interface

**LIRS** Low Inter-reference Recency Set

**LRU** Least Recently Used

**MRU** Most Recently Used

**2Q** 2 Queues

# Bibliography

[ALE$^+$00] M. Aron, J. Liedtke, K. Elphinstone, Y. Park, T. Jaeger, and L. Deller. *The SawMill Framework for Virtual Memory Diversity.* In *Australasian Computer Systems Architecture Conference, Goldcoast, Queensland, Australia*, pages 3–11. IEEE Computer Society Press, 2000. 1, 2

[BM04] S. Bansal and D. S. Modha. *CAR: Clock with adaptive replacement.* Technical report, Standford University, IBM Almaden Research Center, 2004. 9

[Cor69] F. J. Corbató. *A paging experiment with the multics system*, pages 217–228. MIT Press, 1969. 9

[Hel01] C. Helmuth. *Generische Portierung von Linux–Gerätetreiber auf die DROPS–Architektur*, 2001. 11

[HLR$^+$01] C.-J. Haman, J. Löser, L. Reuther, S.Schönberg, J. Wolter, and H. Härtig. *Quality-Assuring Scheduling? Using Stochastic Behavior to Improve Resource Utilisation.* pages 119–128, Dec 2001. 11

[JS94] T. Johnson and D. Shasha. *2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.* Technical report, University of Florida, New York University, 1994. 6, 16

[JZ02] S. Jiang and X. Zhang. *LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance.* Technical report, College of William and Mary, Williamsburg, 2002. 6, 7

[Lie96] J. Liedke. *The L4 reference manual*, Sep 1996.

[Men03] M. Menzer. *Portierung des DROPS Device Driver Environment (DDE) für Linux 2.6 am Beispiel des IDE-Treibers*, Sep 2003. 11

[Men04] M. Menzer. *Entwicklung eines Blockgeräte-Frameworks für DROPS.* Master's thesis, Aug 2004. 11, 32

[MM03] N. Megiddo and D. S. Modha. *ARC: A self-tuning, low overhead replacement cache.* USENIX File and Storage Conference (FAST), Mar 2003. 8, 16, 27, 31

[Reu01] L. Reuther. *DROPS Block Device Driver Interface Specifictaion*, 2001. 11

[srg03] Operating system research group. *L4Env – An environment for L4 applications.* Technical report, University of Technology Dresden, 2003.