Doctoral Thesis

File System Virtualization and Buffer Cache Management in Personal Storage Servers

Woojoong Lee (이 우 중)

Electrical and Computer Engineering

Pohang University of Science and Technology

2011

개인 스토리지 서버 상 파일 시스템 가상화 및 버퍼 캐시 관리

File System Virtualization and Buffer Cache Management in Personal Storage Servers

File System Virtualization and Buffer Cache Management in Personal Storage Servers

by

Woojoong Lee

Electrical and Computer Engineering

(Department of Computer Science and Engineering)

POHANG UNIVERSITY OF SCIENCE AND TECHNOLOGY

A dissertation submitted to the faculty of the Pohang University of Science and Technology in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Division of Electrical and Computer Engineering

Pohang, Korea

Dec 23. 2010.

Approved by

_____

Major Advisor

File System Virtualization and Buffer Cache Management in Personal Storage Servers

Woojoong Lee

The undersigned have examined this dissertation and hereby certify that it is worthy of acceptance for a doctoral degree from POSTECH

12 / 23 / 2010

Committee Chair	박 찬 익 (Chanik Park)
Member	김 종 (Jong Kim)
Member	김 치 하 (Cheeha Kim)
Member	송 황 준 (Hwangjun Song)
Member	이 철 훈 (Cheol-Hoon Lee)

DECE 20053364 이 우 중, Woojoong Lee, File System Virtualization and Buffer Cache Management in Personal Storage Servers, Division of Electrical and Computer Engineering, 2011, 90 p, Advisor: Chanik Park (박찬익). Text in English

Abstract

This thesis studies a file system virtualization technique for unified data management on personal devices. An efficient buffer cache management algorithm is also presented for performance enhancement of the proposed system.

In Part I, we introduce a self-managed distributed file system, named *PosCFS*, which has been proposed for file system virtualization on *Device ensembles* defined as cooperative networks of computing devices in personal area network (PAN). As portable devices are becoming more available and more diverse, people carry number of portable devices such as smart phones, laptops, MP3 players and digital cameras at the same time. In this environment, accessing and managing personal digital contents scattered on each device is really terrible business to users. Every devices has own specialized network or restricted I/O interface. It is not trivially solvable because of not only the heterogeniety of the underlying networks and network file systems on operating systems of the devices, but also the complexity of configurations of them. The rapid growth of personal content accompanied by advances in storage technologies also makes the management process difficult. To solve these challenges, we have proposed the *PosCFS* file system. All the storages in the network could be integrated automatically into a virtual storage by the file system. The file system provides per-user global namespace (*virtual directories*) for managing and accessing data stored on a virtual storage constructed from physical storage spaces detached to device ensembles. The file system was implemented by a peer-to-peer model and using the UPnP protocol to automatically build up a virtual space over all personal devices in the network, and also by using WebDAV for file I/O.

In Part II, we present a second-level buffer cache algorithm for performance enhancement of the proposed file system. Multi-level buffer cache hierarchy is commonly deployed on network file or storage systems. Several studies showed that the I/O access pattern on the second-level buffer cache of file servers or storage controllers, whose reuse distance distribution is hill-shaped, differs from that of the upper-level. This implies that two consecutive accesses to a certain data block have a relatively long temporal distance due to upstream caches while exhibiting weaker temporal locality. The reason is that only missed requests from the upper-level are delivered to the lower-level cache. The Adaptive Replacement Cache (ARC) proposed by IBM dynamically balances recency and frequency using two Least-Recently-Used (LRU) queues in response to changing access patterns. Due to its simplicity, it has low computational overhead compared to other algorithms while performing well across varied workloads. On a second-level cache, however, ARC cannot perform efficiently when the cache size is equal to or smaller than the first-level one. If the reuse distance of most I/O accesses on the second-level cache exceeds its cache size, the recency queue of ARC would not contribute to the cache hits any more. Furthermore, cache hits near the LRU position of the recency queue causes shrinking of the frequency queue as well. In order to solve the problem, we propose a reuse-distance aware block replacement algorithm based on ARC in which the reuse distance pattern is considered by designing the recency queue as a sliding window over a long history buffer for I/O requests. In the proposed algorithm, only blocks within the range of the window are maintained by the cache and the window position is dynamically determined by speculating the minimal reuse distance of a workload. Experimental results show that the proposed algorithm outperforms traditional replacement methods for second-level buffer caches, including Multi-Queue (MQ) as well as ARC with low and constant operating cost. The results also demonstrate that the proposed method efficiently provides a certain degree of exclusive caching like X-RAY without any semantic information of upper-level file systems.

Contents

List of Figures

List of Tables

Part I

File System Virtualization for Unified Data Management on Personal Devices

CHAPTER 1

Introduction

1.1 Background

Nowadays, as portable devices are becoming more available and more diverse, people carry number of portable devices such as smart phones, laptops, MP3 players and digital cameras at the same time. The ensemble of digital devices, a.k.a. the *Device Ensemble* [41] defined as a cooperative network among digital devices in Personal Area Network, is emerging as a new model for personal computing at home, at work, or on the go. For example, schedule information or email messages are synchronized between a smart phone and a desktop PC. Movie clips or photos taken by a camcorder are usually shared among tablet

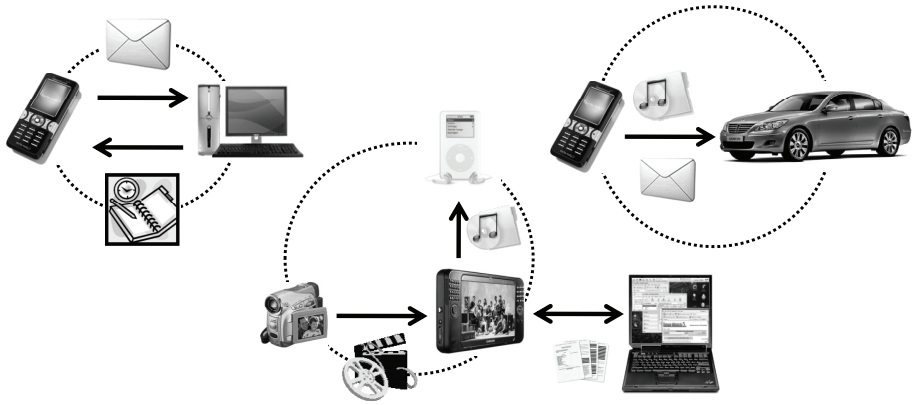


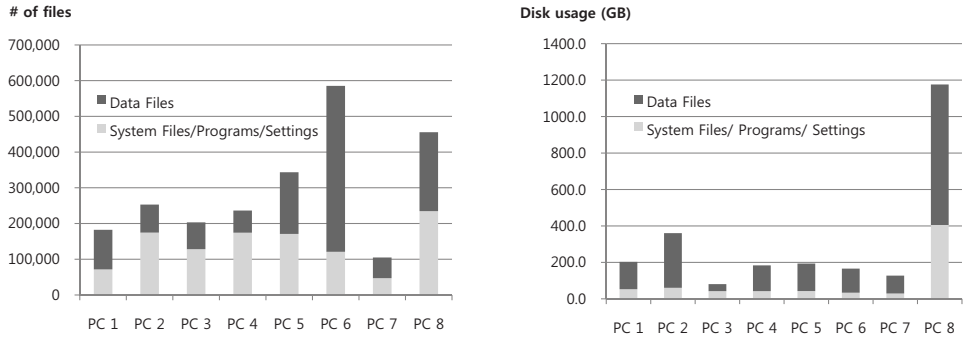
Fig. 1.1 Device Ensembles

PCs or laptops. Document or music files are also copied to one another and played on each device.

In this environment, accessing and managing personal digital contents scattered on each device is really terrible business to users. Every devices has own specialized network or restricted I/O interface. For instance, a cell phone has a USB-type interface for exporting its storage space as a USB mass storage device and a Bluetooth interface for headsets while a smart phone is equipped with WLAN interface. The heterogeneity and ad-hoc characteristics of the *Device Ensemble* makes people harder to manage their digital contents.

Moreover, personal digital contents are rapidly increasing with the recent advance of flash memory and small form factor hard disk technology. The amount of personal content is growing at an enormous rate [23], and according to our investigation for the amount of personal content - this investigation was conducted with PCs and file servers in System Software Lab. at POSTECH in 2010, people usually maintain over 200,000 files on their PC or other personal devices and this is not small scale anymore.

Currently, we have various distributed file systems such as NFS [42], AFS [9],



(a) Number of Files (b) Disk Usage

Fig. 1.2 Personal Contents in Desktop PCs

CODA [40] and Microsoft DFS [27]. However, they are not suitable for device ensembles because they provide only a static mount mechanism and are dedicated to a specific OS platform. Additionally, another critical problem is that most portable devices have limited I/O devices. This affects user's ability to browse data. To provide a convenient data browsing and managing, the traditional directory based file addressing method should be improved.

To sum up, following issues must be handled in a distributed file system that provides unified data management on *Device ensembles*.

- **Heterogeneous underlying networks and OS platform support:** Device ensembles consist of diverse personal devices. The devices are equipped with several wireless interfaces, such as Bluetooth, Wireless LAN (802.11x), UWB or I/O interfaces like USB for data transmission. The operating systems of the devices are also diverse, including Microsoft Windows, Linux, Mac OX, Plam OS, Simbian, etc.

- **Network plug-and-play and self-configuration mechanism support:** Not only the heterogeneity of underlying networks but also ad-hoc characteristics of the networks make people harder to manage their digital con-

tents. The complexity of the network and file system configurations must be eliminated by providing a network plug-and-play and self-configuration mechanism.

- **Scalability and Efficiency:** The file system must be scalable for the rapid growth of personal contents, and mobile device capabilities, wireless network throughput and energy consumption must be considered in the distributed file system.

- **Namespace management and Metadata search for improving data accessibility:** All of the existing file systems have a namespace for file I/O, for instance, directories and file path represent file addresses on most systems. However, the structure is rigid and should be maintained by implicit assignment. For that reason, people tend to forgot where to store their files. This situation may become more serious when they manages enormous amount of files on device ensembles. Unified search functionality for personal content on device ensembles must be provided in order to improve data accessibility. The functionality will be mandatory service for users, context-aware applications and services, i.e. smart meeting room services, or intelligent home media systems, etc.

- **Closed-platform consumer electronic devices (CEDs) support:** In real world condition, most CE devices are closed-platform which means it is not easy or even impossible to install software components like a distributed file system in the CE devices because they are proprietary. Moreover, the data types supported by these CE devices are extremely limited comparing with laptops or desktop PCs. For instance, a Portable Network Graphics (PNG) format image may not be readable on a smart phone. Even if the phone supports the image format, the image resolution may lead to trouble. These limitations must be overcome in the proposed system.

To solve these challenges, we have proposed a self-managed distributed file system, named *PosCFS* [21, 20, 22, 19], which could be adapted to the requirements with storage virtualization. All the storages in the network could be integrated automatically into a virtual storage by the file system. The file system provides per-user global namespace (*virtual directories*) for managing and accessing data stored on the virtual storage. The file system was implemented by using peer-to-peer manner and using the UPnP protocol [11] to automatically build up a virtual space over all personal devices in the network, and also by using WebDAV [6, 45] for file I/O.

File System virtualization in *PosCFS* was represented by two main interfaces. One was an WebDAV-based *storage* interface for I/O access and the other was the *virtual directory* interface for semantic file addressing. The virtual directory trees was dynamically created by matching the file metadata maintained by the file system with keyword-base queries on each client-side. Every file data in *PosCFS* is accessible through Linux VFS-like file I/O interfaces.

The proposed file system also included an event notification framework for managing file and metadata consistency [22] as well as an agent framework for closed-platform CEDs to integrate them into the virtual storage space [19]. An event notification framework is an essential part in self-management of file data distributed across multiple mobile devices in personal area network. Namespace management on file systems, cache management or some other functionalities related to data management inevitably depend on event notification. The proposed framework supports asynchronous event notification for file I/Os and file metadata changes on *PosCFS* service nodes. Since *PosCFS* can provide a unified namespace in all file data of mobile devices distributed in personal area network, the file I/O monitoring functionalities of the event notification service similar to the inotify [24] are easily supported. Moreover, the framework is able to support condition-based metadata monitoring for virtual

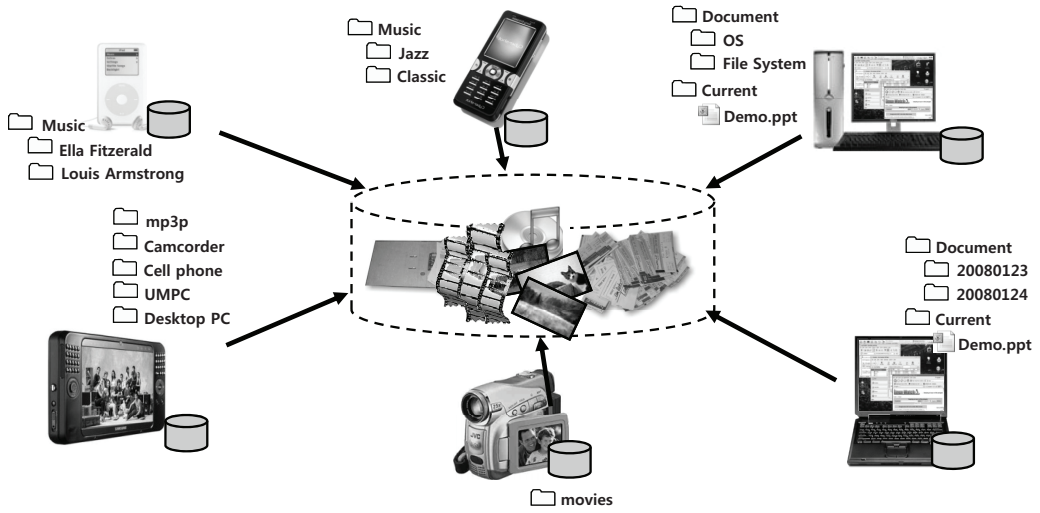


Fig. 1.3 File System Virtualization for Unified Data Management on Personal Devices

directories, which is not supported in the inotify framework of Linux kernel.

The CE device support fuctionality was also included in the file system because, in the real world, we could not install or setup the file system components on CE devices such as digital cameras, MP3 players or smart phones; most of them were closed platform. Moreover, the data types supported by these CE devices are extremely limited comparing with laptops or desktop PCs. For instance, a Portable Network Graphics (PNG) format image may not be readable on a smart phone. Even if the phone supports the image format, the image resolution may lead to trouble. The proposed CE device agent for *PosCFS*, which can be executed on general purpose computers, e.g. laptops, desktop PCs or PDAs with USB-host functionality, etc. Most CE devices can be attached to a host computer and be accessed as a removable USB mass storage device, so that the agent exports the storage space of a CE device on behalf of the passive device. However, it is not simple to provide the virtual directory interface to

the device as if the CE device is a normal *PosCFS*-enabled node because we cannot modify the default file browser or file system on the device. Furthermore, the operation of CE devices, such as taking a photo or playing a music file, may be disabled while it is connected to a host computer. Thus we use a prefetching technique with predefined user profile. Details of the technique will be described later. We also take advantage of the event notification framework proposed previously in [22] for automatic data transcoding in the prefetching process because most CE devices can only support limited data types as we mentioned before.

1.2 Related work

This section describes the state of the art of the smart file systems that provides intelligent and integrated framework for personal content management in PAN.

The GAIA's context-aware file system [7] proposed by the System Software Research Group at University of Illinois was the first approach which tried to adapt a context-aware concept to a file system in the Active Space, an intelligent PAN. It not only provided a novel concept as a well-defined middleware component but also was applicable to diverse computing environments. However, it is not suitable to device ensembles due to its centralized file system construction mechanism, i.e. there must be one mount server for constructing a shared space between devices, and lack of representations for describing file metadata.

The OmniStore [15] proposed by University of Thessaly not only tries to integrate portable and backend storage in PAN, but also exhibits self-organizing behavior through spontaneous device collaboration. Moreover, the system provides transparent remote file access, automated file metadata annotation and a

simple data replication framework. In spite of the innovative features, the system has a limitation in terms of interoperability because it was implemented with its own defined discovery and file I/O protocols rather than standard protocols.

The EnsemBlue [35] proposed by University of Michigan provides a global namespace shared by all of devices in PAN, which is maintained by a centralized file server. It also utilizes energy efficiency and file I/O performance. These features inherited from BlueFS [30] also developed by the same authors. However, it only provides a static and global shared space, a global file tree, among devices that belong to users of the same group, a family or an organization. EnsemBlue also supports closed-platform CEDs with a centralized file server with data synchronization by an event notification framework called persistent query.

Regarding the event notification in distributed file system, the persistent query leverage the cache management system implemented on BlueFS. For delivering events, it utilized a special file for saving reading and writing events and provided a set of event delivering with the file, such as pq_create, pq_delete, and pq_open, etc. When a persistent query created as a file on the server, the events written in the query are automatically delivered to other devices consisting of device ensemble by the cache management scheme. Using the persistent query, it managed the file I/O namespaces of the central BlueFS server as well as CE devices.

Despite the innovative features, in a fully distributed system, e.g., the proposed file system, such a way of persistent query implementation in EnsemBlue does not guarantees delivering events to a particular node because there is no permanently available device like the central server.

UPnP protocol [11] also provides an event notification method based on GENA protocol for state variables on UPnP device. However, the state variables must

be assigned before advertising or discovering, and also be restricted for some device states. Thus, it is hard to leverage the protocol for an event notification framework on the proposed file system.

Several studies related to namespace management were conducted. In our work, we present a concept called per-user global name space which is supported by virtual directories. it provides a semantic namespace inspired from previous researches. In order to support semantic namespace in file service, files can be indexed by their semantic metadata and accessed by the information. SFS [5], LISFS [34], and LiFS [2] addressed the issues of how to generate semantic information and index and access files with the information.

The remainder of Part I is organized as follows. Chapter 2 gives a brief outline and details of the *PosCFS* file system architecture. The event notification framework and the CE device agent framework also presented in Chapter 2. Experimental evaluations are given in Chapter 3. Finally, Chapter 4 presents our conclusion and future work.

CHAPTER 2

PosCFS: A Self-managed Distributed File System

We have proposed a self-managed distributed file system, named *PosCFS* [21, 20, 22, 19], that provides file system virtualization for unified data management on *Device ensembles*.

This chapter introduces the overall architecture of the file system, which is shown in Fig 2.2. *PosCFS* automatically detects every devices available in PAN and integrates the storage space of all the devices into a virtual storage. In order to facilitate data access to the virtual storage, the file system provides per-user global namespace for managing and accessing data on the storage. The namespace in the proposed file system, we named it a virtual directory tree, is constructed based on the file metadata and *user profiles*. The *user profile* can be defined per device or per user and it containes virtual directory creation rules for each of them. More details are discribed in following sections.

2.1 Overall architecture of the file system

The *PosCFS* file system was implemented with two separate components: the client and the service component. *PosCFS* nodes use UPnP [11] protocol to discover and control one another in peer-to-peer model. The WebDAV [6, 45] protocol is used for file I/O in the system. This protocol is an extended version of HTTP, which defines some extended methods for supporting file I/O on a traditional network file system, such as file writing, directory and file property management and locking, as well as the basic methods defined as HTTP, GET, and POST, which are methods for file reading. By using these global standards, we have been able to implement a platform-independent and self-constructible file system.

Linux Virtual File System (VFS) support is done by implementing a bridge component using the Filesystem in Userspace (FUSE) [44] module on Linux kernel. The functionality of the component is simply mapping virtual directory tree managed by PosCFS API to the file tree in Linux VFS; Refer to Fig 2.3.

File system virtualization in *PosCFS* is represented by two main interfaces. One was an WebDAV-based *storage* interface for I/O access and the other was the *virtual directory* interface for semantic file addressing. The virtual directory trees are dynamically created by matching the file metadata maintained by the file system with keyword-base queries on each client-side.

In *PosCFS*, we present a semantic file addressing concept. All of the existing file systems have a namespace for file I/O, for instance, a directory structure represents file addresses on most systems. However, the structure is rigid and should be maintained by implicit assignment. For that reason, people tend to forgot where to store their files. This situation may become more serious on device ensemble or when they manages enormous amount of files. Especially, the explosion of personal data inspire the necessity of a new file addressing mechanism despite of emerging desktop search tools such as Google Desktop

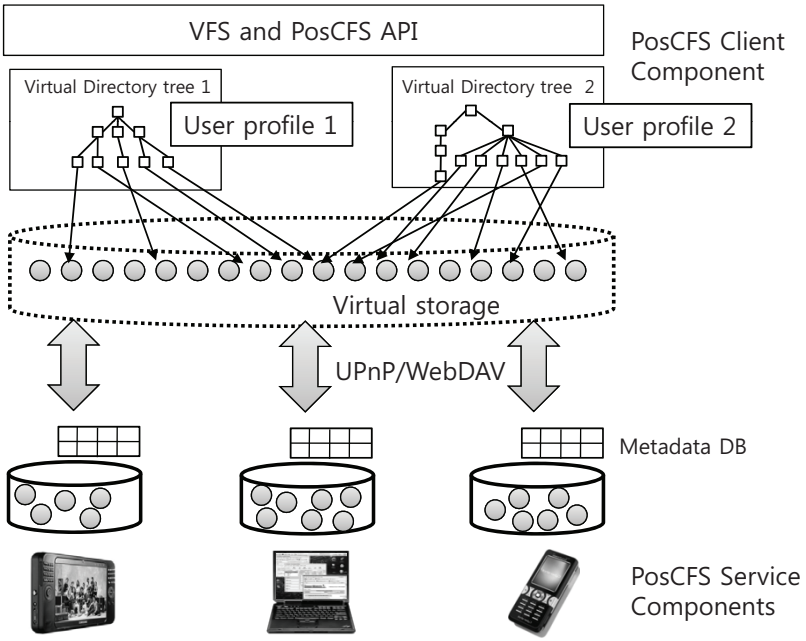


Fig. 2.1 A Conceptual View of File System Virtualization in the *PosCFS* file system: A virtual storage is constructed by peer-to-peer manner in personal area network and the virtual directory tree can be dynamically created on the client-side.

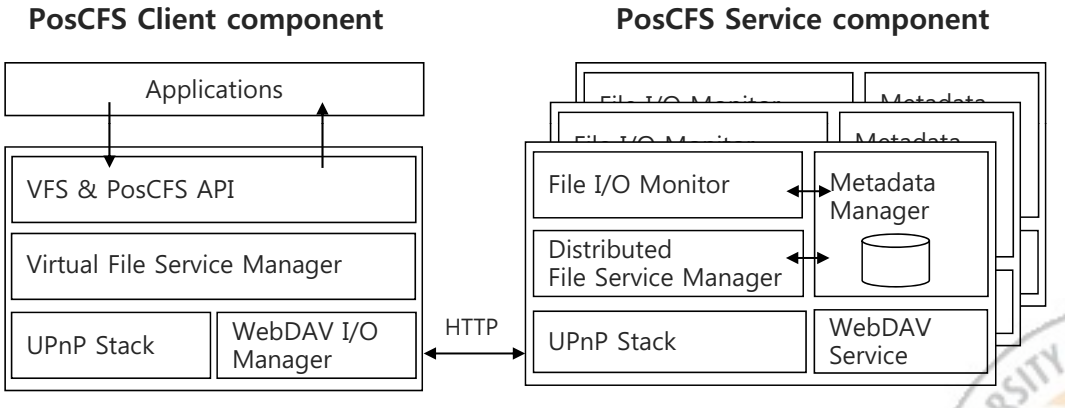


Fig. 2.2 The PosCFS File System Architecture

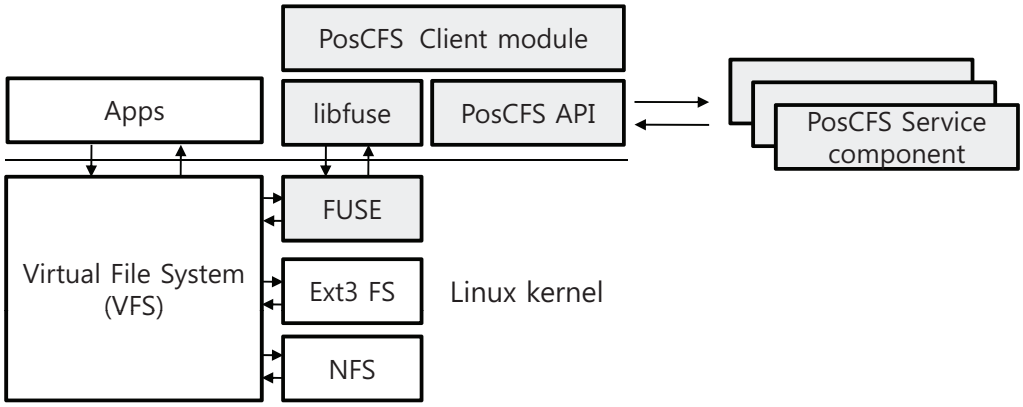


Fig. 2.3 Linux VFS and PosCFS: Filesystem in Userspace (FUSE)

Search [10] and beagle[4].

People usually types several keywords in the search box which representing files that they want to search. This method is also applied to file systems and make them more comfortable for managing their files. For example, we can more easily access a file if we can use semantic metadata as representing file addressing like “title=PosCFS & author=W.Lee & file-type=ppt” rather than the existing address scheme, “/home/wjlee/presentation-file/poscfs.ppt”, because the former is more natural and user-friendly way than the latter when we try to accessing files. Especially, this scheme may be more useful if it can be apply to a distributed file system for integrating storage space on device ensembles.

2.1.1 File addressing method based on user profile

The *virtual directory* is a basic unit of semantic file addressing in our system. A virtual directory tree instance is dynamically constructed by matching the file metadata maintained by the file system with some conditions, such as user queries or profiles managed by the *Virtual File Service Manager* in Fig 2.2.

The profile information consists of two parts. One is the *view preference* that

defines *view-types*; two types of view are provided: *storage view* and *virtual directory view*, and the other is *virtual directory construction rules*.

Below, we briefly describe the virtual directory construction rules with some examples. The rules could be described with some pre-defined commands, such as 'DIR(NAME | NAMING-RULE) {CONDITIONS}' and 'SDIR(NAME | NAMING-RULE) {CONDITIONS}', and some specific keywords.

From the sequence of *DIR* and *SDIR* the categorization rules are included by some specific condition that correspond to the file metadata and context information maintained by the PosCFS service component. For instance, the case 'a)' in the example means, creating a virtual directory named 'docs' without sub-directories and with the condition, all the files in the directory should be document file class. Whereas the case 'b)' in the example means creating the virtual directory with sub-directories where each of them is named with distinctive author values.

The rules can be simply converted into the Structured Query Language (SQL) in RDBMS with some other information such as the current user context. (e.g., user location, events, etc.)

a) DIR(docs){file-class="document"};

b) DIR(docs){file-class="document"};SDIR(author=*){};

c) DIR(music){file-class="music"};SDIR(artist=*){};

d) DIR(wjlee){file-class="document" & author="wjlee"}

e) DIR(file-class=*){}; SDIR(location=*){};

f) DIR(current){ctx-name="project meeting"};

g) DIR(snapshot){}; SDIR(ctime=*){ctime $\leq$ 20070505 & ctime $\geq$ 20070501};

These rules are maintained by each client node and virtual directories can be dynamically constructed by querying these requests to service nodes.

2.1.2 Service profile

The *user profile* was used for constructing virtual directory tree as we mentioned in section 2.1.1 while the *service profile* is mandatory information for initialization of the *PosCFS* service component, which are composed of two parts. One is the basic service profile that is defined and advertised by the SSDP/UPnP and the other is the extended service profile which is exported to other devices by an UPnP action defined in the file system. Since CE devices cannot manage their virtual directory tree with the profile, A PosCFS agent service handles it on behalf of the devices, i.e. it provides PosCFS client functionalities for the device; we will discuss about details of the agent framework later.

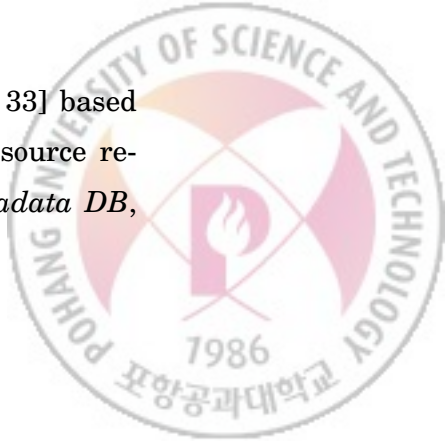
1) **Basic Service Profile :** all the properties of the profile are defined in UPnP specification; *Device Type*, *Friendly Name*, *Manufacturer URL*, *Model Description*, *Model Name*, *Model Number*, *Model URL*, *Serial Number*, *Universal Device Name(UDN)*, and *Universal Product Code(UPC)*

2) **Extended Service Profile**

- *Storage Size* : The storage size of the device. This field can be used to regulate prefetching operation.
- *Repository Path* : represents an export directory.
- *Supported data types* : mime type list delimited by ';'
- *Optional fields for data types* : optional fields, which can be used for describing details of the supported data types. e.g. resolution:320x240, etc.

2.1.3 Metadata management

In order to maintain file metadata, we implemented a SQLite [8, 33] based *metadata repository* which enables fast metadata retrieving on resource restricted devices. The roles of the *metadata repository*, a.k.a. *metadata DB*,



File Table

uri (file_id)	fname	type	size	ownership	permission

Attribute Table

id	attribute	value	uri

Fig. 2.4 The File and Attribute Table in Metadata DB : The Universal Resource Identifier (URI) is represents a file reference in WebDAV protocol.

managed by the *Metadata manager* module in the service component are to store and manage the semantic metadata of files. The *metadata DB* is managed by a background process which carries out monitoring of file modification in order to extract semantic information from each file and to annotate the file with the extracted information. Refer to Fig 2.5.

Most file formats have their own metadata fields. For instance, in the case of the MP3 format, the *file I/O monitor* extracts some semantic information, such as *Artist* and *Genre* from the ID3 tag of the format. The pdf or other document formats have some attributes like *Author*, *Title*, or *Subject*, and so on. For extensibility, the DB table had been designed, whose internal representation was similar to the RDF triple structure [17], resulting in no limitation of the number of attribute-value pairs attached to a file resource.

User-annotated metadata as well as mandatory file metadata, such as file name, type, size, ownership, and access permission, is also used for indexing files. For example, context information related to a user schedule or event can be represented with some keywords like locations, date and times, or schedule names, etc.

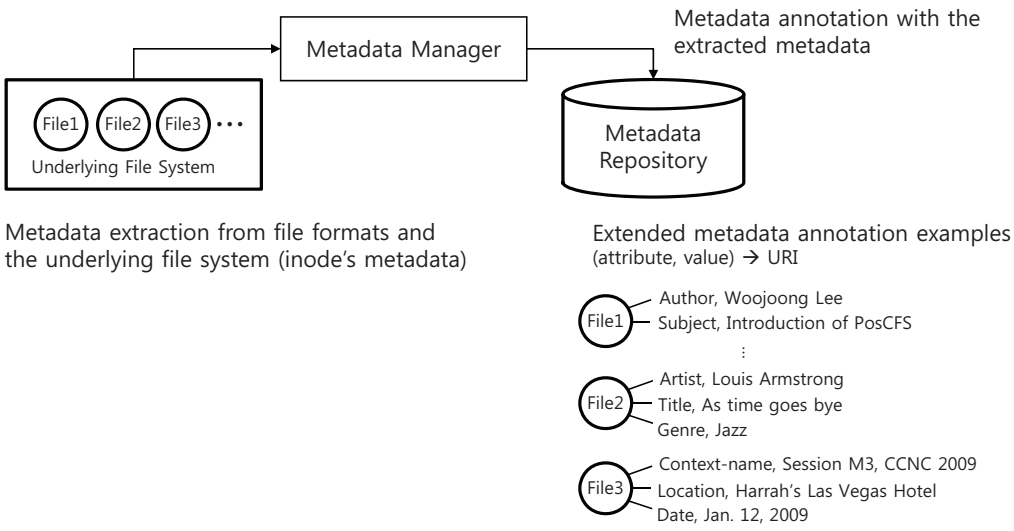


Fig. 2.5 Metadata Annotation Process

2.2 An Event notification framework

2.2.1 Introduction

This section present the design and implementation of an event notification framework for *PosCFS* file system. An event notification framework is an essential part in selfmanagement of file data distributed across multiple mobile devices in personal area network. Namespace management on file systems, cache management or some other functionalities related to data management inevitably depend on event notification.

The proposed framework supports asynchronous event notification for file I/Os and file metadata changes on *PosCFS* service nodes. Since *PosCFS* can provide a unified namespace in all file data of mobile devices distributed in personal area network, the file I/O monitoring functionalities of the event no-

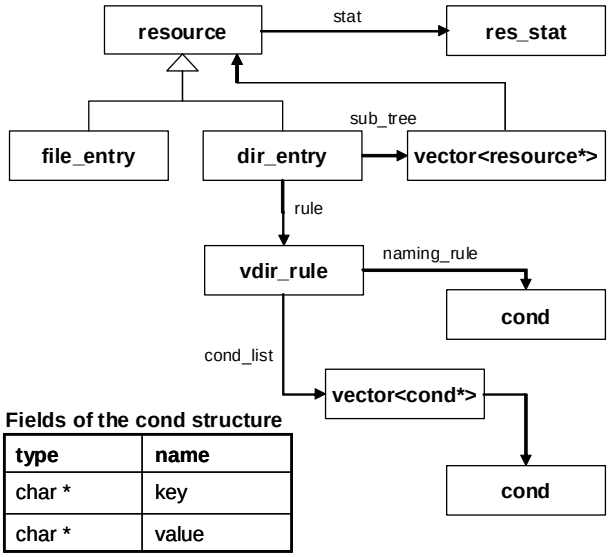


Fig. 2.6 Resource, File_entry, Dir_entry and Cond Structures

tification service similar to the inotify [24] are easily supported. Moreover, the framework is able to support condition-based metadata monitoring for virtual directories, which is not supported in the inotify framework of Linux kernel.

The inotify [24] is available in Linux local file system. It has been included in the mainline Linux kernel since the release 2.6.13. However, the inotify framework could not be directly applied to *PosCFS* since the system design considers only local file systems. This section describe how we designed and implemented an event notification framework for *PosCFS*.

2.2.2 Event notification API

The inotify subsystem provides three functions for event monitoring on user-level: *inotify_init()*, *inotify_add_watch()*, and *inotify_rm_watch()*. Table 2.2 shows the API functions that are extended from the original functions. All the arguments and their types in the prototypes are same to original functions except

Table 2.1 Mask Values defined in Event notification API

Mask Values	Description
IN_ACCESS	A file or directory was read from
IN_MODIFY	A file of directory was written to
IN_ATTRIB	Metadata of a file of directory was changed
IN_OPEN	A file of directory was opened
IN_CLOSE	A file of directory was closed
IN_DELETE	A file of directory was deleted
IN_CREATE	A file of directory was created
IN_DELETE_SELF	A file of directory monitored was deleted
IN_MOVED_FROM	A file of directory was moved away from watch
IN_MOVED_TO	A file of directory was moved to

for the *resource* parameter of the *cfs_inotify_add_watch* function. The original function has the file path on behalf of the *resource*. However, the path information is not enough to describe files or directories in *PosCFS* file system because all the namespace entries on it was constructed by matching semantic queries with file system metadata.

The *resource* class is an abstract class for the *file_entry* and the *dir_entry*, basic units consisting of virtual directory tree on the file system. Figure 2.6 presents the structure and relation of classes related to the *resource* class. The *res_stat* includes attributes of *file_entry* and *dir_entry* for mandatory file I/O: *size*, *c_time*, *m_time*, *a_time*, etc.

Most of all, the *vdir_rule* is a core structure for maintaining the conditions on the *dir_entry* creation time. The *naming-rule* and *cond_list* fields of the structure are filled with the *virtual directory construction rules* previously mentioned on the previous section 2.1.1.

The conditions maintained by *resource* and the *mask* values of the *cfs_inotify_*

Table 2.2 Event notification API

API name	Description
int cfs_inotify_init();	creates an cfs_inotify interface
*int cfs_inotify_add_watch (int fd, resource *res, int mask);*	starts watching the resource for events defined in mask or cond_list in the resource class, and returns a watch descriptor
int cfs_inotify_rm_watch (int fd, int wd);	stops watching the resource for events

add_watch represent events that application want to monitor. For instance, if a virtual directory was constructed with 'file-class=Music & artist=Ella Fitzgerald', then the *dir_entry* could be updated when a new mp3 file that met the specified conditions was stored on the virtualized storage space on PAN.

The *mask* values defined in our system specify events related to the *resource*, and we leverage the values on the inotify on the Linux system. Some of the mask values are shown in table 2.1. For more details, refer to [24].

Table 2.2 describes the prototypes of API functions provided for applications. The *cfs_inotify_init()* are used to initialize the inotify interface on client-side. When the initialization process is completed application can start or stop watching resources for events defined in mask or conditions with *cfs_inotify_add_watch* or *cfs_inotify_rm_watch* respectively, and the *fd* can be used for an interface to receive events with *read()* or *select()* functions.

2.2.3 Implementation of the event service

The architecture of the event service on *PosCFS* is shown in Fig 2.7. Every service node has two separated repository spaces. In our system, files are stored in local file system and the metadata are managed in *metadata DB* as being

extracted when storing or creating the files on the service node.

The *event manager* component unifies the interface for registering notification request as well as for monitoring events from the underlying repositories using the inotify system and registering hooking functions to the *metadata DB* that was implemented with SQLite3 embedded database system[8].

All the requests for monitoring particular resources are accepted by the *notification service* on service nodes. When receiving monitoring requests from a client node it delegate the requests to a *per-client event notifier thread* that are dynamically created on run-time, and then the designated thread open a connection to the client to deliver events corresponding to the monitoring requests represented by either *mask* or *cond_list* or both of them, at one and the same time, it also registers the requests to the *event manager* in order to detect the target resource changes.

An asynchronous event message delivered to client nodes includes the specified fields in table 2.3. The message format is exactly identical to the event structure defined in inotify system.

The event notification framework was applied to the VFS-support module of *PosCFS*. The module was implemented with FUSE module [44] in Linux kernel in order to support native applications on Linux system. The functionality of the module is simply mapping virtual directory tree managed by PosCFS API to the file tree in Linux VFS.

As applying the event notification framework to the module, the virtual directory tree can be managed more effectively by handling the update event from service nodes.

This framework can also apply to implementing a proxy service which makes closed-platform consumer electronic devices(CEDs) as first-class node of *PosCFS*. Intelligent data management using the event notification framework, e.g. automatic data backup and sync system, can alleviate the poor accessibility of file data on CEDs.

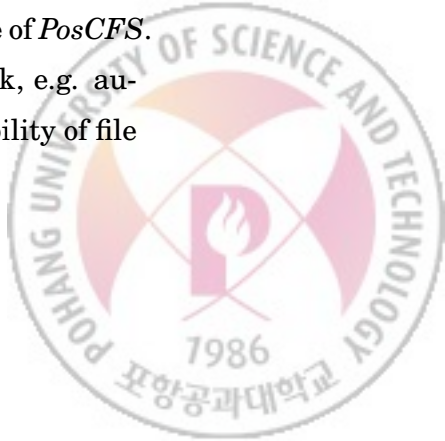


Table 2.3 Event message format

Identifier	Description
int wd	watch descriptor
uint32_t mask	*mask* contains bits that describe the event that occurred
uint32_t cookie	*cookie* is a unique integer that connects related event
uint32_t len	size of *name* field
char name[]	resource name

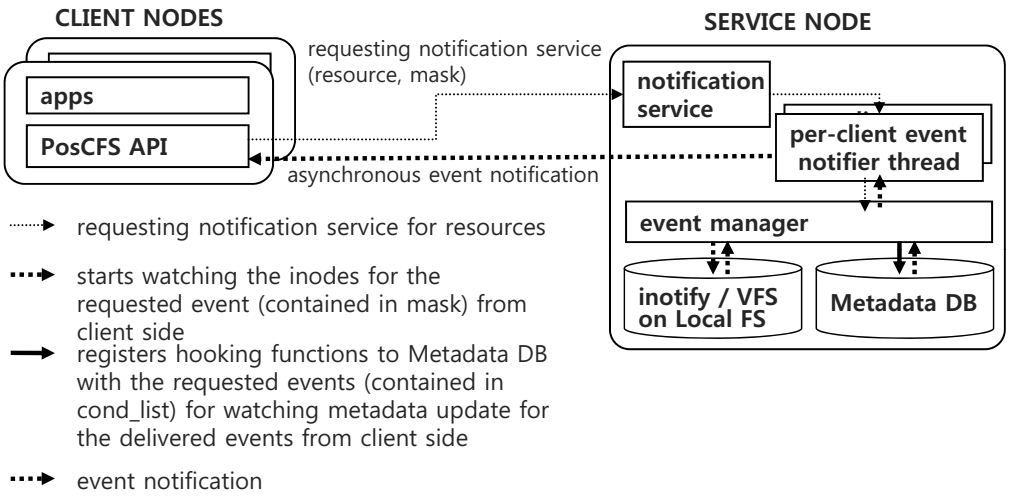


Fig. 2.7: Overview of the Event Notification Framework

2.3 An Agent framework for CE Devices

2.3.1 Introduction

In previous sections, we presented the *PosCFS* file system which automatically detects every devices available in PAN and integrates the storage space of all the devices into a virtual storage. In order to facilitate data access to the virtual storage, PosCFS creates virtual directories which enable a semantic file addressing from applications.

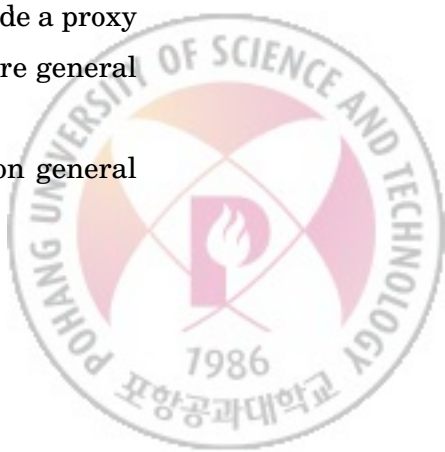
However, ina real world consisting of multiple consumer electronics (CE) devices, it is not easy or even impossible to install software components like PosCFS in the CE devices because they are proprietary.

This section presents the design and implementation of a CE device agent which enables the file system virtualization by *PosCFS* on CE device ensembles. Most CE devices are connected to a host computer via USB interface, so the agent will be executed on the host computer to export the physical storage space of a CE device to *PosCFS*. Then, virtual storages and virtual directories are created by *PosCFS* according to predefined user profile. Finally, to solve the problem of limited data type and format supported in a CE device, the event notification framework proposed in section 2.2 is used for automatic data transcoding.

2.3.2 Design and implementation of the agent framework

This section presents the design and implementation of the CE device agent for *PosCFS*. The major functionalities of the agent framework is to provide a proxy service which makes closed-platform CE devices operate as if they are general *PosCFS-enabled* devices.

As we mentioned before, the agent framework can be executed on general



purpose computers, e.g. laptops, desktop PCs or PDAs with USB-host functionality, etc. Most CE devices can be attached to a host computer and be accessed as a removable USB mass storage device, so the agent exports the storage space of a CE device on behalf of the passive device.

However, it is not simple to provide the virtual directory interface to the device because we cannot modify the default file browser or file system on the device. Furthermore, the operation of CE devices, such as taking a photo or playing a music file, may be disabled while it is connected to a host computer. Thus we use a prefetching technique with predefined user profile.

The issues related to the automatic data transcoding are also important because the data types supported by these CE devices are extremely limited comparing with laptops or desktop PCs. For instance, a Portable Network Graphics (PNG) format image may not be readable on a smart phone. Even if the phone supports the image format, the image resolution may lead to trouble. Thus we take advantage of the event notification framework described in the previous section. The details of this feature will be mentioned later, with introducing the data prefetching technique.

The agent framework utilize the HAL[47] and the D-Bus[38] architecture on Linux for detecting CE device connections and disconnection. When it detected a CE device connection, the default *callouts* which are programs that run on device add/remove, are invoked by the HAL daemon and the event can be sent to the agent framework by the D-Bus protocol. Then the agent inspects the mounted storage space of the CE device whether a special information called *PosCFS service profile* exists or not. The *service profile* will be described in section 2.1.2. If the file is exist, the agent checks the profile and initializes a *PosCFS* service component for the device. Otherwise, the agent creates a new service profile from the service profile template and user inputs.

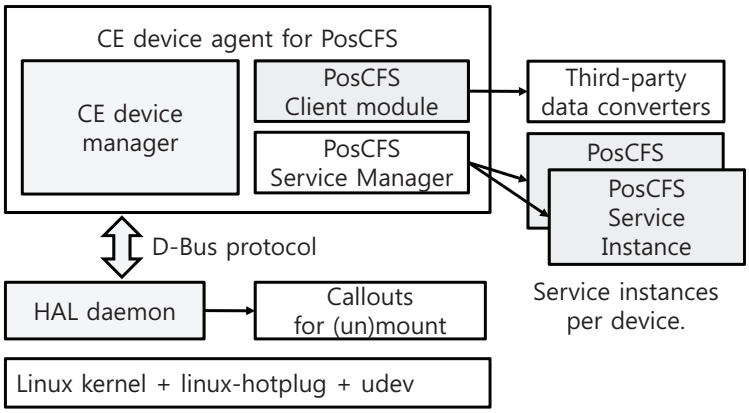


Fig. 2.8 The CE Device Agent Framework

2.3.3 Data prefetching and automatic data transcoding

Most CE devices are closed platform so that we cannot modify or alter the default file browser application or file system on the devices. Furthermore, the operation of CE devices may be disabled while it is connected to a host computer.

In order to provide the virtual directory interface to a CE device, the CE device agent has to directly manage the storage spaces mounted on the host computer. It builds up a virtual directory tree by creating or deleting real directories on the mounted space. The rules for virtual directories can be generated from the *user profile* previously defined by a user. The *user profile* is stored to the CE device's storage space like the service profile. All the data which have to be in the virtual directories are prefetched into the CE device storage in order to make the embedded file browser in the CE device access the data after detaching the CE device.

Some examples of the user profile for the CE device are listed below.

- DIR(Music){mime-types="audio/mpeg"};
- DIR(Music){mime-types="audio/mpeg"}; SDIR(Ella){artist="Ella Fitzerald")};

This is a very useful functionality of *PosCFS*. For instance, let us imagine a situation where an user wants to copy some music files of an artist from a home server to his MP3 player connected by his laptop. First of all, the user must mount each storage of the server and the MP3 player, and he have to find the files on the server. And finally, the user can copy them to the player. However, in *PosCFS*, the series of process described above can be performed very simply and easily. The user only need to plug the player into an available USB port on the laptop and then to insert some conditions for creating a virtual directory tree.

The automatic data transcoding is one of the most important features because the data types supported by these CE devices are extremely limited comparing with laptops or desktop PCs. For instance, a Portable Network Graphics (PNG) format image may not be readable on a smart phone. Even if the phone supports the image format, the image resolution may lead to trouble. However, there are diverse data format, so the file system or service cannot handle all of them. Furthermore, some formats are only encoded by a special program.

Thus we take advantage of the event notification framework which was designed and implemented for managing the virtual directory namespaces so that the all the file system events, especially file metadata changes, could be delivered to client-side.

Some third-party data converting applications like *imagemagick*, *ogg2mp3* or *mp32ogg* can be used for data type converting process and invoked by the agent service using the framework. The agent service usually monitors the virtual storage space. When a file is prefetched into the CE device, the service detects the events and checks the information included in the *service profile* such as *Supported data types* and *Optional fields for data types*. If the file type is not described in the *Supported data types*, then the agent converts the data format to the desired data format using the third-party applications.

CHAPTER 3

Evaluation

In order to evaluate *PosCFS*, we conducted several experiments, particularly on the device discovery time including the initialization time of a virtual storage. For that purpose, a prototype of *PosCFS* was developed in Linux 2.6.12 with Intel UPnP SDK v1.4, SQLite v3.2.7, libneon 0.25.3, and libextrator 0.5.14. The prototype implementation also includes a VFS support module in Linux. We implemented the module using the file system in user-space (FUSE) functionality provided by the Linux kernel and libfuse 2.6.0.

3.1 Micro benchmark

3.1.1 Device discovery and query processing time

The testing platforms we used include four Jaurus-SL5500 PDAs with Strong ARM 266 MHz , 32 MB memory, and 802.11b wireless networks; three PCs with 0.8 GHz to 2.8 GHz Pentium, 512 to 1000 MB RAM, and 100 MB wired Ethernet; and two laptop computers with 1.8 GHz P4, 512 MB RAM, and 802.11g wireless Ethernet. Fig 3.1 shows the results for device discovery time. Note that the device discovery time is not dependent on the number of devices; it gradually converges to 7 seconds. This is because the UPnP discovery process depends on the search time of the simple service Discovery Protocol (SSDP).

The response time to a request for virtual directories by a client is referred to as query processing time which is one of important criteria for scalability. Fig 3.1 also shows how query processing time varies as the number of devices increases. In this evaluation, 10,000 files are uniformly distributed to all nodes. The results demonstrate that the query processing overhead is affordable and little sensitive to the number of devices, but greatly depends on the performance of service nodes; note that the processing overhead is evidently increased proportionally to the number of PDAs.

Query processing overheads for directory depth and number of files are evaluated in Fig 3.3. The depth is determined by the virtual directory constrcution rules which described in Section 2.1.1. The results shows that the processing overhead is quite sensitive to the directory depth. The reason is that the limitation of the metadata indexing structure; in order to guarantee extensibility for extended metadata indexing, the proposed file system maintains *(attribute, value)* and *file object* mapping on the metadata DB. However, the indexing structure requires large memory footprint because the query processing requires multiple *JOIN* [16] operations proportioned to the number of attribute

keywords which was used for virtual directory creation; in future work, this limitation should be considered for performance enhancement.

3.1.2 File I/O performance

In order to evaluate the file I/O performance and the CE Agent framework of the proposed file system, we performed the IOzone filesystem benchmark [31]. The testing platforms we have used include a Sony VAIO S55LP laptop with Pentium M, 1GB RAM, 802.11g, a Fujitsu P1510 mini-tablet with Pentium M, 512MB RAM, 802.11g and a Nikon D-80 digital camera. The average initialization time of the proxy service invoked by the CE device agent was about a second, and the file I/O performance of the proxy service for CE devices was shown in Fig 3.4.

The read performance of *PosCFS* was slightly better than or comparable to the NFS-user; the write performance was affected by the metadata annotation process. The performance of the proxy service, however, was slightly lower than that of NFS userserver. Especially, the write performance of the agent were quite poor than that of other file systems due to slow write speed of the digital camera and the data transcoding effect.

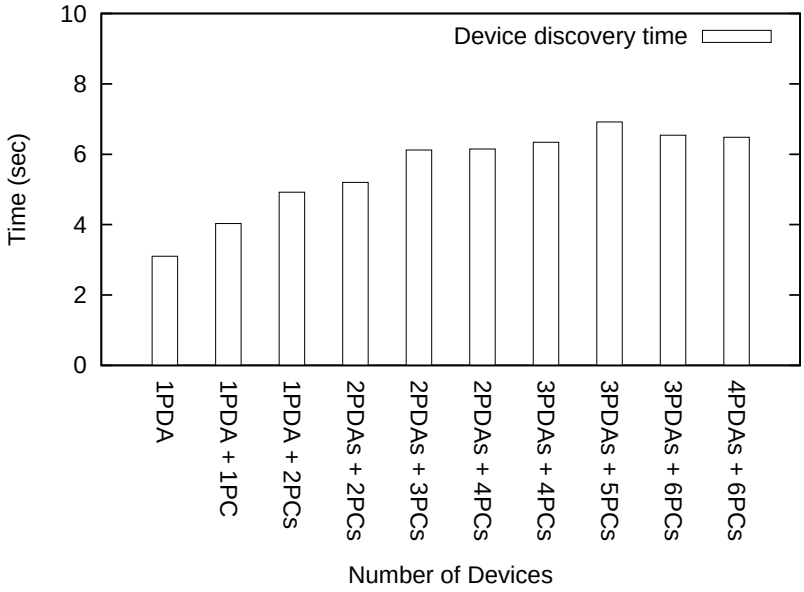


Fig. 3.1 Device Discovery Time

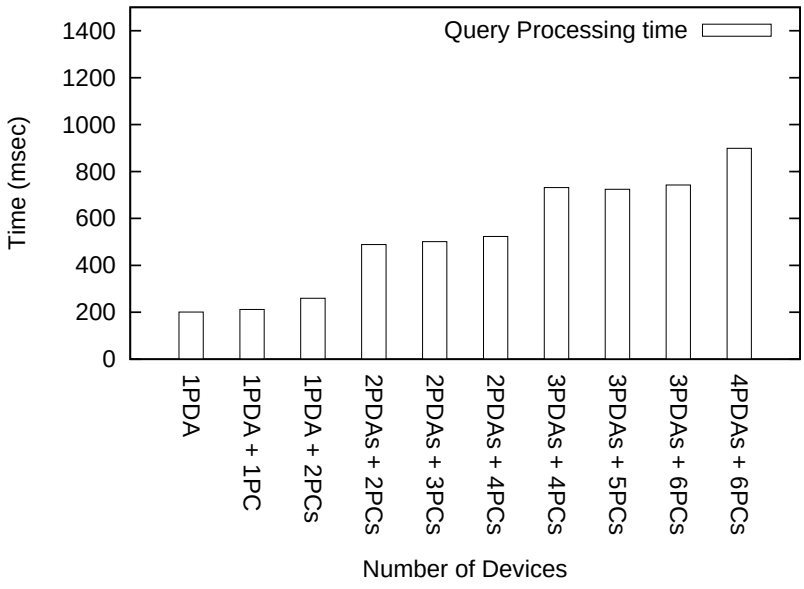


Fig. 3.2 Query Processing Time

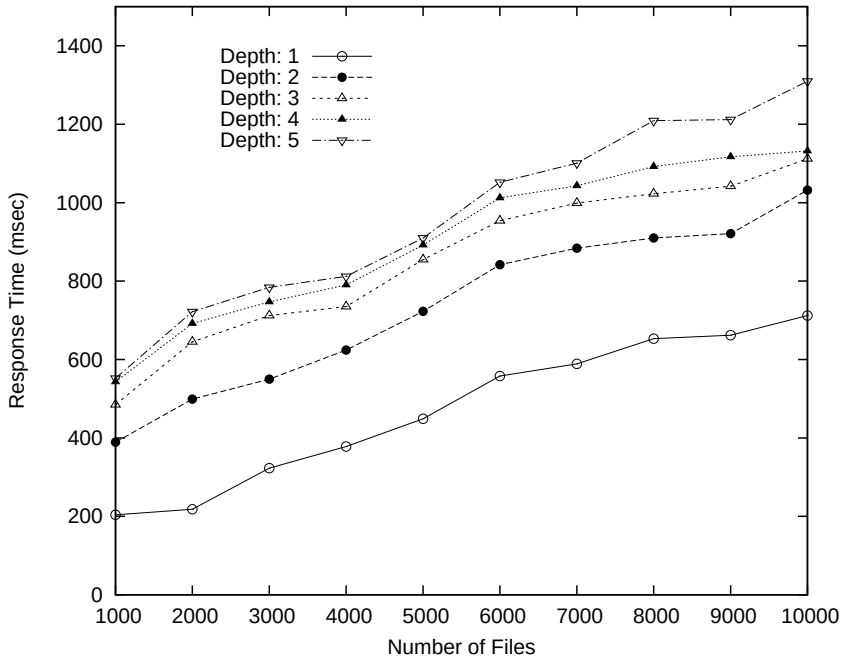


Fig. 3.3 Virtual Directory Creation: Query processing overheads for directory depth and number of files

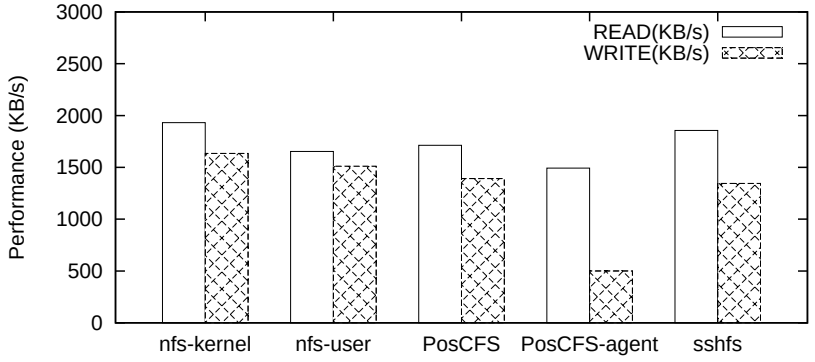


Fig. 3.4 IOzone Benchmark Results: Comparison of File I/O Performance of Network File Systems

CHAPTER 4

Conclusion and Future Work

In Part I, we introduce a self-managed distributed file system, named *PosCFS*, that provides file system virtualization for unified data management on personal devices. All the storages in the network could be integrated automatically into a virtual storage by the file system which provides per-user global namespace (*virtual directories*) for managing and accessing data stored on the virtual storage. The file system was implemented by a peer-to-peer model and using the UPnP protocol to automatically build up a virtual space over all personal devices in the network, and also by using WebDAV for file I/O.

The proposed file system supports asynchronous event notification for file I/Os and file metadata changes on *PosCFS* service nodes. Since *PosCFS* can provide a unified namespace in all file data of mobile devices distributed in personal area network, the file I/O monitoring functionalities of the event noti-

fication service similar to the inotify are easily supported. Moreover, the framework is able to support condition-based metadata monitoring for virtual directories, which is not supported in the inotify framework of Linux kernel.

The CE device support fuctionality was also included in the file system. The agent can be executed on on general purpose computers. Most CE devices can be attached to a host computer and be accessed as a removable USB mass storage device, so that the agent exports the storage space of a CE device on behalf of the passive device. However, it is not simple to provide the virtual directory interface to the device as if the CE device is a normal *PosCFS*-enabled node because we cannot modify the default file browser or file system on the device. Furthermore, the operation of CE devices, such as taking a photo or playing a music file, may be disabled while it is connected to a host computer. Thus a prefetching technique was proposed in order to solve the problem. The event notification framework was also utilized for automatic data transcoding in the prefetching process because most CE devices can only support limited data types.

In future work, more extensive evaluation with realistic workloads must be performed. Privacy and security should be also considered as well as efficient data indexing and caching for performance enhancement. In Part II, we present a second-level buffer cache algorithm for performance enhancement of the proposed file system.

Part II

Reuse Distance Aware Block Management in Adaptive Replacement Cache

CHAPTER 5

Introduction

5.1 Background

Buffer caches are used to enhance the performance of file or storage systems by reducing I/O requests to underlying storage media. In particular, multi-level buffer cache hierarchy is commonly deployed on network file systems or storage systems. In this environment, the I/O access pattern on second-level buffer caches of file servers or storage controllers differs from that of upper-level file systems.

The reuse distance of a block is an important metric to characterize the cache access behavior. It is defined as the number of requests between two adjacent accesses to a block address in an I/O stream. The access pattern on second-level buffer caches analyzed and studied by Zhou et al. [48, 49] has a hill-shaped

reuse-distance distribution. This implies that two consecutive accesses to a data block have a relatively long temporal distance due to up-stream caches. They also examined the behavior of the access patterns in terms of frequency, and revealed that the more frequently the block is accessed, the larger portion it takes out of total accesses.

Fig 5.2 and 5.3 show reuse-distance distributions on each hierarchical layer and how first-level cache size affects I/O patterns on second-level buffer caches. On the first-level cache, more than 95 percent of correlated accesses have a reuse distance less than or equal to 256. Whereas most accesses on second-level have a reuse distance larger than 2K or 4K, depending on the size of the first-level cache. The reason for the weak temporal locality on second-level is that only missed requests on the upper-level are delivered to the lower-level cache. In other words, we can simply consider the first-level cache a request filter where only block requests with the reuse distance smaller than the cache size are maintained by the cache. Both cases in Fig 5.3 present the hill-shaped reuse distance distribution where the beginning, peak, and end of the "hill" region are designated as *minimal distance*, *peak distance* and *maximal distance*, respectively, and the *minimal* and *peak distance* are determined by the size and the hit ratio of the first-level cache.

Various techniques including frequency-based block prioritizing [48, 49], exclusiveness [3, 46], or multi-level cache coordination [12], have been proposed with consideration of the temporal or frequency pattern of I/O workload on the multi-level cache hierarchy. However, the complexity issue remains still unsatisfactory.

The Adaptive Replace Cache (ARC) algorithm proposed by Megiddo et al. [25, 26] dynamically balances recency and frequency by using two *Least-Recently-Used* (LRU) queues in response to changing access patterns. ARC is simple to implement and has low computational overhead while performing well across varied workloads.

ARC not only outperforms most online algorithms, such as LRU, *Frequency-Based Replacement* (FBR) [39], *Least-Frequently-Used* (LFU) [1], and *Low Inter-reference Recency Set* (LIRS) [13], but is also comparable to offline algorithms LRU-2 [32], *2-Queue* (2Q) [14], and *Least-Recently/Frequently-Used* (LRFU) [18].

However, because ARC does not take into account the reuse distance of I/O requests, ARC cannot perform efficiently on a second-level cache. Note that the reuse distances of most I/O accesses on the second-level cache will be long so that the recency queue of ARC cannot contribute much to the cache hits. Furthermore, due to the long reuse distance of I/O requests on the second-level cache, most cache hits will be observed near the LRU (not MRU) position of the recency queue in ARC. If cache hits are observed near the LRU position of the recency queue, ARC tries to increase the size of the recency queue in order to capture the recency locality more. Accordingly, the size of the frequency queue

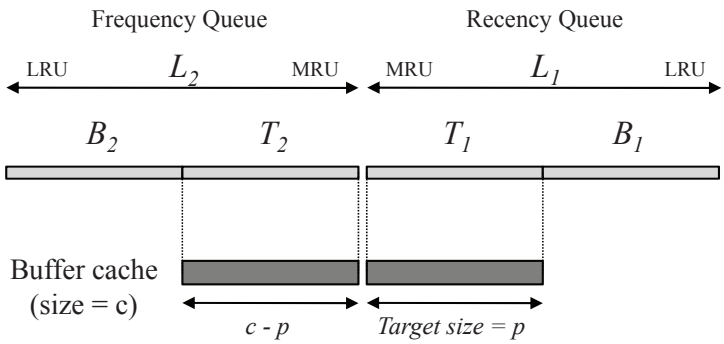


Fig. 5.1 The Adaptive Replacement Cache algorithm: ARC maintains two variable-size LRU queues L_1 and L_2. L_1 maintains blocks that have been accessed only once, recently, while L_2 maintains blocks that have been accessed more than twice. Each queue is divided into two parts, top and bottom queues, and bottom queues are ghost buffers which just keep block numbers evicted from each top queue; i.e., L_1 is comprised of T_1(top) and B_1(bottom), and L_2 is comprised of T_2(top) and B_2(bottom), respectively. Refer to the algorithm 5 in Appendix A

decreases. It becomes worse when the second-level cache size is equal to or smaller than the first-level cache size. In this case, the recency queue of ARC contributes nothing to the cache hits.

In order to solve the problem, we propose an enhanced block replacement algorithm, called RARC (Reuse-distance aware ARC). In order to capture the reuse distance of I/O requests, a history buffer over all I/O requests is maintained. A sliding window of concern on the history buffer will determine which I/O blocks are kept in the recency queue. So, the size of the sliding window is equal to the size of the recency queue in RARC and the sliding window will be dynamically updated reflecting the reuse distance pattern on the second-level cache.

The main challenge in RARC is to capture the reuse distance pattern in order to control the sliding window to maintain I/O blocks of high demand in the cache. Among the reuse distance pattern, the minimal reuse distance plays a key role because the upper-level cache size usually keeps changing for various reasons like memory pressure. The history buffer will be used to estimate the minimal reuse distance.

The remainder of Part II is organized as follows. A comparison with related work is provided in Section 5.2. Section 5.3 gives workload characteristics on second-level buffer caches and describes the problem of the ARC algorithm. In Chapter 6, we present details of the RARC block replacement policy and the reuse-distance aware block replacement. Experimental evaluations are given in Chapter 7. Finally, Chapter 8 presents our conclusions and future works.

5.2 Related work

The *Multi-Queue* (MQ) algorithm [48, 49] was proposed in order to adapt second-level workload characteristics. The method prioritizes block accesses with their frequency, and maintains them using multiple LRU queues. Each block is maintained with its reference count, and placed at a proper priority-level LRU queue. The temporal characteristic of the second-level was considered with the *expiration time* of a block in order to demote a block from a higher to lower priority queue. Even if MQ has constant time overhead, it requires the expiration time scaning for all LRU blocks of queues, and the scanning has to be done on every request. Therefore, the algorithm has relatively higher overhead than other online replacement policies such as LRU, ARC etc.

Exclusive caching is another technique for multi-level buffer caching where buffer caches on each level maintain exclusive data blocks in order to avoid the duplication of data at different levels of the cache hierarchy. Since only block addresses are observed by the second-level cache, it is not trivial to keep data blocks exclusively on each layer because the upper-level buffer cache size may continuously change. In [46], Wong and Wilkes solve this problem by adding a new interface between each layer. They propose a new SCSI command DEMOTE which moves a block from the file system buffer cache directly to the underlying storage cache.

On the other hand, X-RAY [3] proposed by Bairavasundaram et al. exploits file system semantic information in order to predict the size change in the upper-level file system buffer cache and maintain the exclusive block set on the second-level storage cache. The file system semantic information is retrieved by tracking the arrival frequency of I/O requests to identify whether they are data blocks or inode block - usually inode blocks are periodically synchronized to storage media every few seconds. The monitoring system also investigates inode block writes in order to extract some semantic information like a file ac-

cess time and which data blocks belong to the file. With these activities, the underlying storage layer can speculate which blocks are cached in the upper-level file system buffer cache. As a result, X-RAY has better exclusiveness in cache than MQ. However, it requires excessive I/O monitoring overhead, and in some cases, e.g., if a drastic cache size change occurred during a certain time interval, massive background block reads will be issued to underlying storages in order to fetch the exclusive data blocks into the storage cache.

5.3 Workload characteristics and the problem of the ARC algorithm

In this section, we clarify workload characteristics on multi-level cache hierarchy and how the first-level cache size affects I/O patterns on second-level buffer caches that were previously studied by Zhou et al.[48, 49] We also present the problem of the ARC algorithm more specifically with some run-time examples.

5.3.1 Workload characteristics on second-level buffer caches

In order to demonstrate the workload characteristics, we implement an LRU simulator for an upper-level cache and extract second-level I/O traces possibly observed by a second-level cache. The LRU cache sizes of 4K and 8K entries are considered, and the 253.perlbmk-diffmail workload trace from SPEC 2000 benchmark [37], collected by the Performance Evaluation Laboratory at Brigham Young University, is used as the input stream of the simulation.

Reuse distance distribution on each layer is shown in Fig 5.2 and 5.3. The hill-shaped distribution on the second-level appears in Fig 5.3-(a), (b), and they also demonstrate that the *minimal* and *peak distance* depends more on the size and hit ratio of the first-level cache, whereas the *maximal distance* depends more on workload characteristics.

Note that the hit ratio of the first-level cache affects the reuse distance distribution observed by the second-level cache.

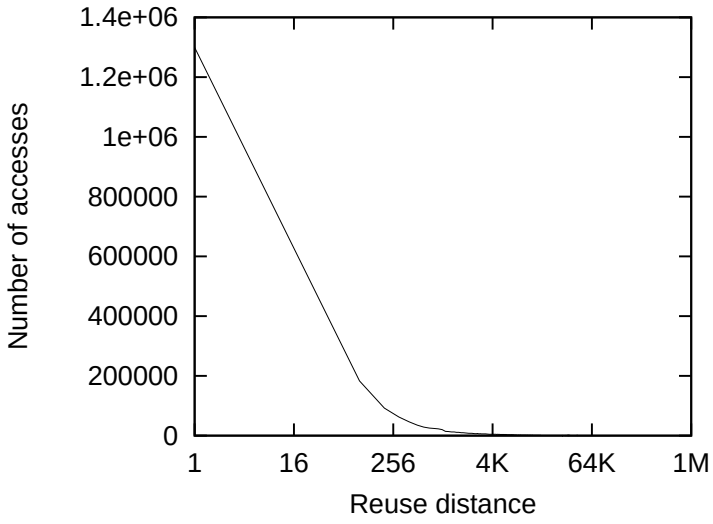
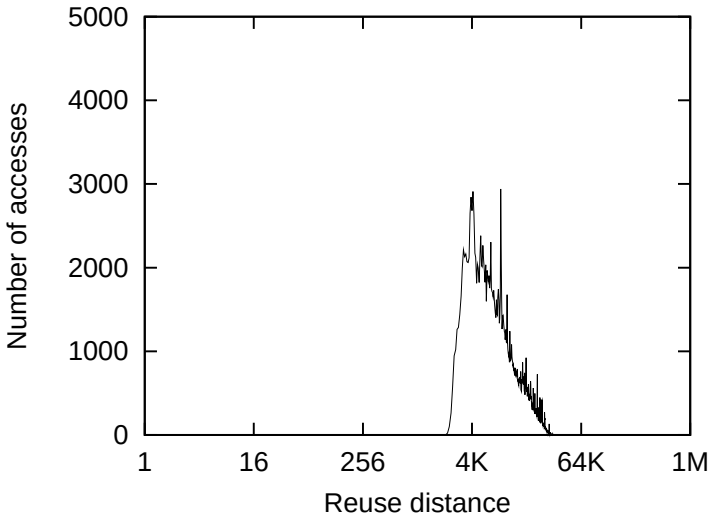


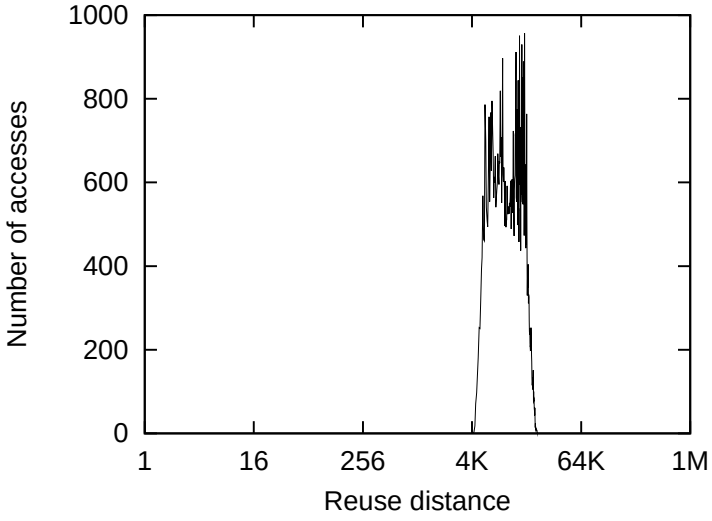
Fig. 5.2 Reuse distance pattern on the first-level cache: The 253.perlbmk-diffmail workload trace from SPEC 2000 benchmark was used as the input stream of the simulation. The result shows that more than 95 percent of correlated accesses have a reuse distance less than or equal to 256.

5.3.2 Brief review of the ARC algorithm

As we mentioned in the previous section, the Adaptive Replacement Cache (ARC) algorithm does not consider the second-level workload characteristics, that is, reuse distance pattern. It just balances recency and frequency by using two LRU queues in response to changing access patterns. ARC maintains two variable-size LRU queues L_1 and L_2. L_1 maintains blocks that have been accessed only once, recently, while L_2 maintains blocks that have been accessed more than twice. Each queue is divided into two parts, top and bottom queues, and bottom queues are ghost buffers which just keep block numbers evicted



(a) Reuse distance pattern on the second-level cache: the first-level cache size is set to 4K entries



(b) Reuse distance pattern on the second-level cache: the first-level cache size is set to 8K entries

Fig. 5.3 Reuse distance pattern on the second-level caches where the beginning, peak, and end of the "hill" region are designated as *minimal*, *peak* and *maximal distance*, respectively. The *peak distance* of (a) is about 4K and the *minimal distance* is about 2K, while the *peak distance* of (b) is about 8K and the *minimal distance* is about 4K.

from each top queue; i.e., L_1 is comprised of T_1(top) and B_1(bottom), and L_2 is comprised of T_2(top) and B_2(bottom), respectively.

Let c be the cache size. ARC maintains each queue with the following rules.

- $0 \leq |L_1| + |L_2| = |T_1| + |T_2| + |B_1| + |B_2| \leq 2c$
- $0 \leq |T_1| + |B_1| \leq c$
- $0 \leq |T_1| + |T_2| \leq c$
- New entry is inserted into the MRU position in T_1.
- Any entry in T_1, T_2, B_1, and B_2 that gets referenced is moved to the MRU position in T_2.

Initially, all queues, T_1, T_2, B_1, and B_2 are set to empty, and a tunable parameter p is set to 0. The p is an integer ranging from between 0 to c that indicates target size of T_1; ARC attempts to keep p most recent blocks in L_1 and $c - p$ most recent blocks accessed more than twice in L_2.

Therefore, a victim block is selected from L_1 when $|T_1| > p$ or ($|T_1| = p$ and requested block $b_x \in B_1$); the LRU entry is evicted from T_1 and inserted into the MRU position in B_1. In the other case, the victim is selected from L_2; the LRU entry is evicted from T_2 and inserted into the MRU position in B_2.

In order to capture both recency and frequency of workloads, ARC controls the parameter p dynamically. Cache hits in B_1 make p increase because it means the reuse-distances of recently accessed blocks are larger than current T_1 size; if the size of T_1 increases by up to p, then block accesses in the current workload can be hit in T_1. Whereas hits in B_2 make p decrease because T_2 requires more spaces, as the number of blocks accessed more than twice increases in the current workload; it makes the size of T_2 larger.

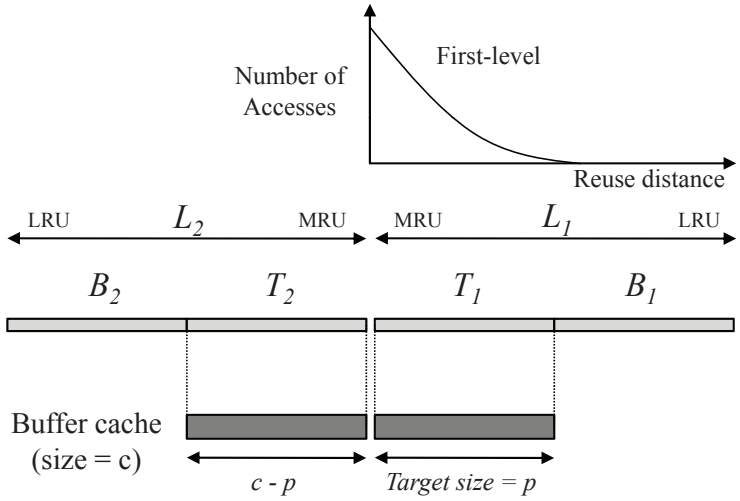
For more details, please refer to [25, 26] and algorithms 5, 6 and 7 in Appendix A.

5.3.3 The problem of the ARC algorithm

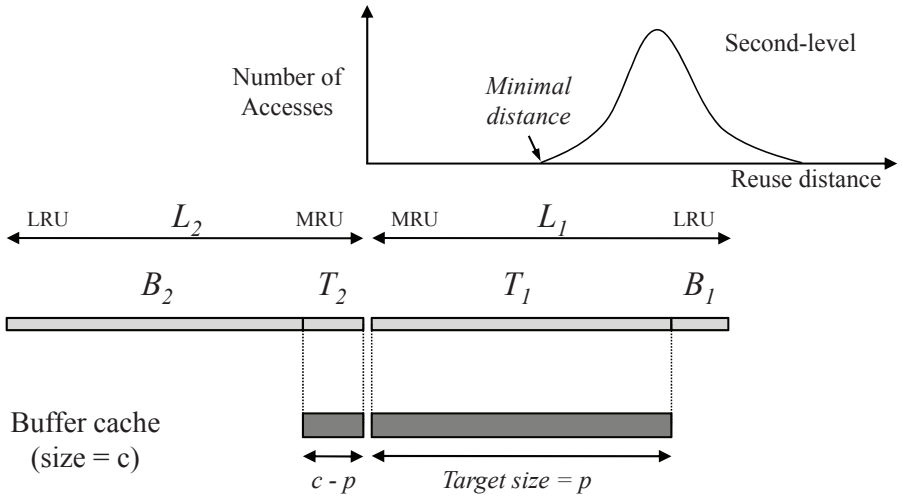
ARC is simple to implement and has low computational overhead compared to other algorithms while performing well across varied workloads. Moreover, the efficiency of ARC for second-level workloads is comparable with MQ policy, and in some cases, it outperforms the algorithm.

However, the caching efficiency can be decreased because ARC does not consider the second-level workload characteristic. To put it concretely, ARC cannot perform efficiently when the cache size is equal to or smaller than the first-level one. In that case, the reuse distance of most I/O accesses on the second-level cache will exceed its cache size so that the recency queue of ARC will rarely contribute to the cache hits. Furthermore, cache hits near the LRU position of the recency queue cause shrinking of the frequency queue as well.

The effect on a series of this process is demonstrated in Fig 5.4-(a) and (b). Fig 5.4-fig3-a shows the state of a buffer cache based on the ARC algorithm in first-level cache. Most block accesses have a reuse distance smaller than the cache size so that the size of T_1 and T_2 can be adjusted efficiently in response to workload changes. On the other hand, in the case of Fig 5.4-(b), T_1 is extended by the weak temporal locality of second-level I/O accesses while T_2 is reduced by control rules of ARC; the cache size is smaller than the *peak distance* of the workload so that hits in B_1 are increased compared to the former case. Another side effect is only a small portion of the accesses will be captured and moved to T_2; this make the hit ratio in T_2 worse along with the reduced size of T_2.



(a) ARC on first-level cache



(b) ARC on second-level cache

Fig. 5.4 The problem of the ARC algorithm

CHAPTER 6

Reuse-distance aware Adaptive Replacement Cache Algorithm

In this section, we present the RARC algorithm that is designed based on ARC. The proposed algorithm solves the problem of ARC and provides adaptability for the I/O pattern on second-level buffer caches.

In the following subsections, we firstly describe the intuition of the proposed RARC algorithm, and then present the details of its queue and buffer cache management.

6.1 Intuition of RARC algorithm

We notice that blocks ranging between the minimal distance of the workload and the MRU of L_1 in the original ARC never contribute to the cache hit ratio as well as they are the main culprit for deterioration of the L_2 effectiveness. To remedy the situation, we extend the ARC algorithm with a long history buffer and a sliding window composed of T_1 and B_1 queues.

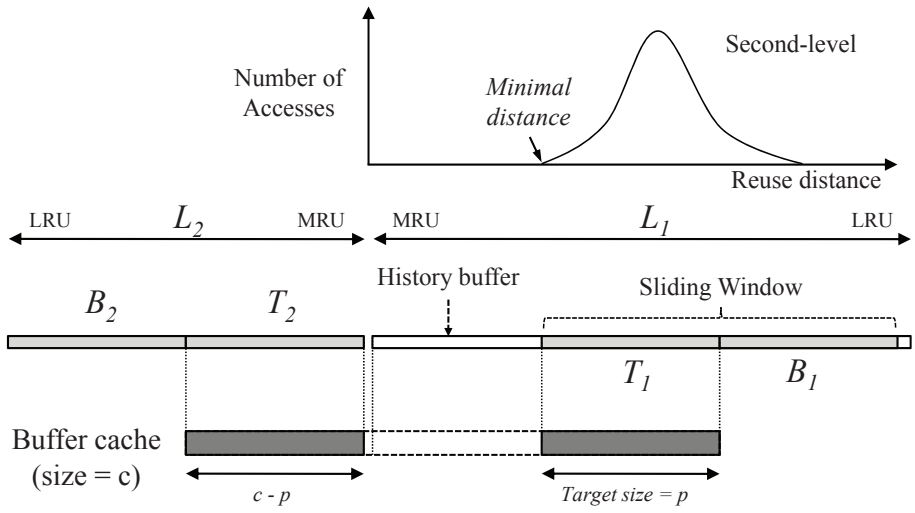


Fig. 6.1 The intuition of the proposed algorithm

Fig 6.1 demonstrates the intuition of the proposed algorithm; compare with Fig 5.4-(b). The newly issued request is not retained by the recency queue, i.e., T_1, but inserted into the MRU position of the history buffer that only maintains the block number and its access time. The recency queue is defined as a sliding window where only blocks within the range of the window are loaded into the cache via the background I/Os. If the sliding window is fixed at the MRU position of the history buffer, then RARC behaves in the same manner with ARC.

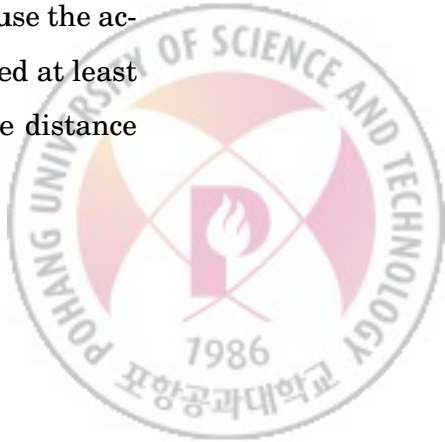
The window position is dynamically adjusted by speculating the minimal reuse distance of a workload. This is another challenging issue because of the frequent size changes of up-stream buffer caches as we mentioned. In order to respond effectively, the algorithm keeps track of the reuse distance of individual blocks using the history buffer and a global clock. This technique also has constant time overhead per request, and requires few CPU cycles. It will be more specifically discussed in the following subsections.

6.2 Details of the proposed algorithm

Before explaining the RARC algorithm, we define the I/O workload on the second-level more formally. To facilitate estimating the minimal reuse distance under the workload where the upper-level cache size keeps changing, we divide the entire workload into several sub-workloads. So, a distinct minimal reuse distance can be found in each sub-workload. Moreover, the window size must be carefully determined because temporal locality can hardly be captured when the window size is smaller than the width of the ”hill” region, i.e., the value of subtracting the *minimal distance* from the *maximal distance*.

We define a workload W as a union of sub-workloads, W_i, where $0 \leq i \leq \xi$ and ξ is a positive integer; each sub-workload has a distinctive *minimal distance* of adjacent workloads, W_{i-1} and W_{i+1}. Note that each sub-workload has a distinct minimal reuse distance.

To estimate the minimal reuse distance for each sub-workload, the *expected reuse distance* is defined for each block, b_t, in the history buffer. Because the actual reuse distance of a block is determined when the block is accessed at least twice, we define the expected reuse distance of a block as the reuse distance assuming that the block is accessed right now.



Algorithm 1: RARC, $cache\_size = c$

Input: The request stream $b_1, b_2, \ldots, b_t, \ldots$
Init. : Set $p \leftarrow 0$, $q \leftarrow 0$, $d \leftarrow 0$, and g_clock$\leftarrow 0$,
and set the LRU lists HB, T_1, B_1, T_2, and B_2 to empty

For every $t \geq 1$ and any b_t,

g_clock $\leftarrow$ g_clock +1;

if b_t is in HB. **then** ADJUST_WINDOW (b_t, d, q);

if b_t is in T_1 or T_2. **then**
Move b_t to MRU position in T_2.
(also remove it from HB if it was in T_1)
if $q > 0$, then CHECK_WBOUNDARY (d, q);

else if b_t is in B_1. **then**
Update $p \leftarrow$ min $(p + \delta_1, c)$;
where δ_1 = $\{1$, (if $|B_1| \geq |B_2|$) or $\lfloor |B_2|/|B_1| \rfloor$, (otherwise) $\}$
REPLACE (b_t, p);
Move b_t from B_1 to the MRU position in $T2$.
(also remove it from HB)
Fetch b_t to the cache, and if $q > 0$, then CHECK_WBOUNDARY (d, q);

else if b_t is in B_2. **then**
Update $p \leftarrow$ max $(p - \delta_2, 0)$;
where δ_2 = $\{1$, (if $|B_2| \geq |B_1|$) or $\lfloor |B_1|/|B_2| \rfloor$, (otherwise) $\}$
REPLACE (b_t, p);
Move b_t from B_2 to the MRU position in $T2$.
Fetch b_t to the cache, and if $q > 0$, then CHECK_WBOUNDARY (d, q);

else
// b_t is not in $T_1 \cup B_1 \cup T_2 \cup B_2$.
UPDATE_HB (b_t);
CHECH_SIZE_AND_REPLACE (b_t, p);
if $q = 0$ **then**
Fetch b_t to the cache, and insert it into the MRU position in T_1.
else
CHECK_WBOUNDARY (d, q);
Fetch b_t and return it.
end

Algorithm 2: UPDATE_HB(b_t, p)

if b_t is in HB - ($T_1 \cup B_1$). **then**
  b_t.clock $\leftarrow$ g_clock;
  Move b_t to the MRU position in HB.
else
  if |HB| $= m$ **then** Delete the LRU page in HB.
  b_t.clock $\leftarrow$ g_clock;
  Insert b_t to the MRU position in HB.
end

Fig 6.2 shows the data structure of RARC. The L_1 queue for capturing the temporal pattern is composed of a history buffer, HB, and a sliding window, W that is divided into two parts, T_1 and B_1. The L_2 queue for frequency is same as that of ARC;

Let c be the buffer cache size, and the size of HB be the m, where $m \geq c$. The history buffer only maintains requested block numbers and the virtual time of each request issued. In order to capture the access time of blocks, a global clock, g_{clock} is managed by the algorithm and incremented by 1 on every read request so that an expected reuse distance of a block request can be simply calculated by subtracting the clock time of the block from the current clock time; for example, let t_c be the current clock. For a block, b_i, which is located at HB[q] and whose requested time is t_i, the expected reuse distance of b_i is $t_c - t_i$. (Refer to the sub-procedure, `get_erd()` in Algorithm 3.)

RARC controls each queue size, i.e., HB, T_1, T_2, B_1, and B_2, with the following rules, which are equivalent to the original ARC algorithm except for the window control.

- $|HB| = m$, where $m \geq c$
- $0 \leq |L_1| + |L_2| = |T_1| + |T_2| + |B_1| + |B_2| \leq 2c$
- $0 \leq |T_1| + |B_1| \leq c$
- $0 \leq |T_1| + |T_2| \leq c$

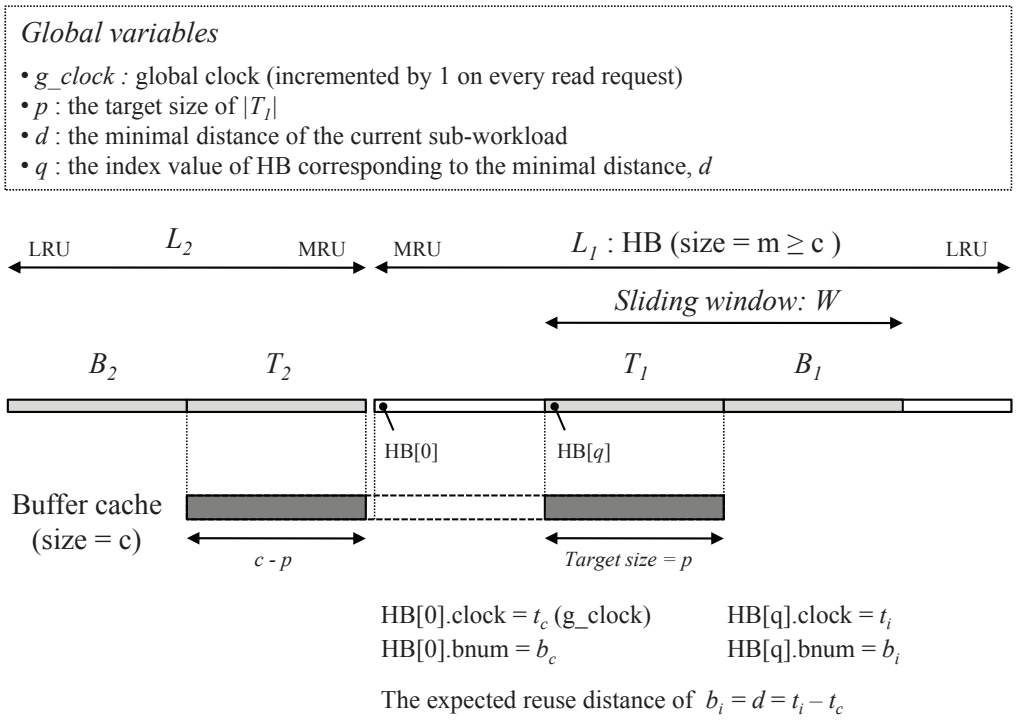


Fig. 6.2: Data structures of the RARC algorithm: The L_1 queue for capturing the temporal pattern is composed of a history buffer, HB, and a sliding window, W that is divided into two parts, T_1 and B_1. The L_2 queue for frequency is the same as that of ARC.

The window, T_1 and B_1, is dynamically adjusted and controlled with the parameter q over the history buffer. The parameter q is an index value of HB where the expected reuse-distance of the block pointed by q is defining the *minimal distance* of the current workload, W_i.

The details of RARC are described in Algorithm 1. Initially, all queues, HB, T_1, T_2, B_1, and B_2, are set to empty and global variables, g_clock, p, q and d are set to 0. The d parameter indicates the *minimal distance* of the current workload, and q is an index value of HB corresponding to d.

For every request stream of W_i (b_1, b_2, ..., b_t,...), the algorithm speculates

whether or not the *minimal distance* is to be changed. This is done by the ADJUST_WINDOW procedure in Algorithm 4 and the process will be presented later in this section. Secondly, the algorithm classifies and handles each request according to the following four cases.

These four cases are identical to the original ARC except that blocks within the range of T_1 are retained in the buffer cache by the background I/O.

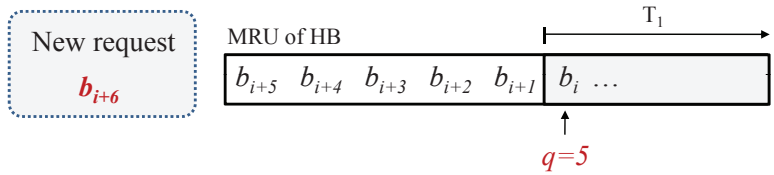
- Case I (cache hit): If a block request, b_t, is in T_1 or T_2; The referenced block is moved to the MRU position in T_2. Then the expected reuse distance of the block $HB[q-1]$ is examined in order to decide whether or not the start boundary of the window should be moved. Note that, this test is only activated when $q > 0$. If the expected reuse distance of that block is equal to or larger than d, then the block data is fetched into the buffer cache at the MRU position in T_1; the q is decremented by 1 as well. In the other case, i.e., the expected reused distance is smaller than d, no additional operation is required. This window boundary management process is described in Algorithm 3 - the procedure CHECK_WBOUNDARY, and Fig 6.3.

- Case II (cache miss): If a block request, b_t, is in B_1; The parameter p which denotes the target size of T_1 is incremented by 1 (if $|B_1| \geq |B_2|$) or by $\lfloor |B_2|/|B_1| \rfloor$ (otherwise) up to the cache size, c. This p control is identical to the original ARC. Then, in order to update the buffer cache and T_2, a block eviction process is performed by the REPLACE procedure; for more details of the procedure, refer to algorithm 6 in Appendix A. Lastly, the window boundary management described in the previous case is performed as well.

- Case III (cache miss): If a block request, b_t, is in B_2, all the process is same as the B_1 hit case except for the p control. The p is decremented by 1 (if $|B_2| \geq |B_1|$) or by $\lfloor |B_1|/|B_2| \rfloor$ (otherwise) up to 0.

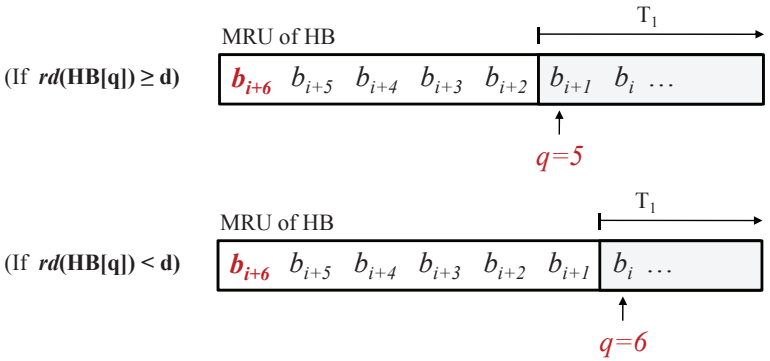
- Case IV (cache miss): If a block request, b_t, is not included in any of T_1, T_2, B_1, and B_2, the block is inserted into the MRU position in HB; if the block is found in HB, it is just moved to the position. Block eviction as well as queue size control are performed by the procedure `REPLACE` and by `CHECK_SIZE_AND_REPLACE` before updating HB and T_1. Lastly, the window boundary management is done by the procedure `CHECK_WBOUNDARY` where the block data of $HB[q]$ is fetched into the buffer cache when the expected reuse distance of $HB[q]$ is equal to or larger than d; otherwise, the q parameter is incremented by 1. Note that every block in HB is shifted by 1 toward the LRU position due to the newly inserted block, b_t; the block data of $HB[q]$ is not in the buffer cache.

Fig 6.3 gives some examples of the boundary management where we assume two cases; one is that a newly requested block does not reside in any queues and the other is the opposite case. In the case of Fig 6.3-(a), the block b_{i+1} is included in the range of T_1 due to the newly inserted b_{i+5}; the block HB[q], i.e. b_{i+1}, should be tested to determine whether its expected reuse distance is equal to or larger than d. If it is true, block data of b_{i+1} is fetched into the buffer cache, otherwise, the window is moved to the left by 1; q is incremented from 5 to 6. In the case Fig 6.3-(b), there is no change in the history buffer after the insertion; the expected reuse-distance of the block HB[$q-1$], i.e. b_{i+1}, is tested for the window boundary update. If the condition is met, block data is fetched and q is decremented from 5 to 4. Otherwise, no additional operation is required. The details of this process are described in Algorithm 3, the procedure `CHECK_WBOUNDARY`.



If b_{i+6} is not in $T_1 \cup T_2 \cup B_1 \cup B_2$:

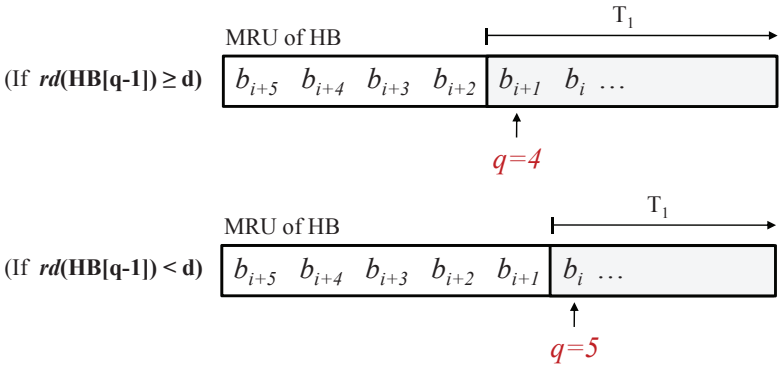
· b_{i+6} is inserted into the MRU position in HB
· Every block is shifted by 1 toward the LRU position



(a) A new request is inserted into the MRU position in HB

If b_{i+6} is in $T_1 \cup T_2 \cup B_1 \cup B_2$:

· b_{i+6} is moved to the MRU position in T_2
· no index value is changed in HB, but the expected reuse distance of each block is incremented by 1



(b) A cache hit occured, no change in HB

Fig. 6.3 Examples of the window boundary management

Algorithm 3: CHECK_WBOUNDARY (d, q)

if HB has been updated with a new request **then**
  if `get_erd` (HB[q]) $\geq d$ **then**
    `CHECH_SIZE_AND_REPLACE` (HB[q], p);
    Fetch HB[q] to the cache and insert it to the MRU position in T_1
  else
    $q \leftarrow q + 1$;
else
  if $q > 1$ *and* `get_erd` (HB[q-1]) $\geq d$ **then**
    $q \leftarrow q - 1$;
    `CHECH_SIZE_AND_REPLACE` (HB[q], p);
    Fetch HB[q] to the cache and insert it to the MRU position in T_1
end

Subroutine: `get_erd` (x)
  `// returns the expected reuse distance of the request` x
  return g_clock - x.clock

6.3 Window control

In this section, we present a minimal distance speculation method for window control in the RARC algorithm. As we mentioned before, the minimal distance of workload continuously changes due to variations of the upper-level read cache size.

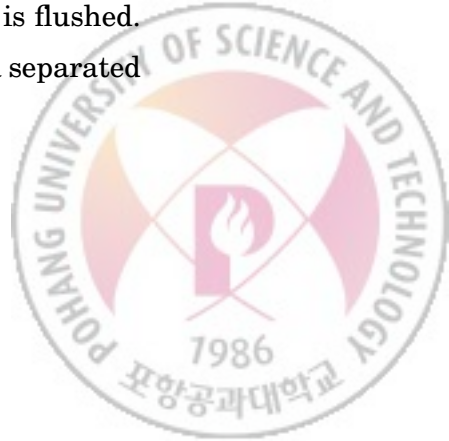
In order to estimate the minimal distance, the RARC algorithm keeps track of the reuse distance of each request with the history buffer. Fig 6.4 and Algorithm 4 describe the process of the calculation performed by the procedure `ADJUST_WINDOW`.

In this algorithm, a new sub-workload is detected when reuse distances of all I/O requests are not in the range of the current window during a test period. The test period is defined as the number of I/O requests, thus describing a transition state between sub-workloads, e.g., a transition W_i to W_{i+1}. If n consecutive I/O requests are detected to have larger reuse distance than the

current minimal reuse distance given a test period of n, a new sub-workload is detected and the algorithm sets a new minimal reuse distance to be the minimal one among the n consecutive I/O requests. However, if the reuse distance of a new request is smaller than the current minimal reuse distance, the algorithm immediately sets a new minimal reuse distance to the reuse distance of the new request without waiting for a test period; refer to the example in Fig 6.4.

The parameter q is set to a corresponding value of the minimal reuse distance (denoted as d), i.e., the index value of the block selected in the minimal distance estimation process. For example, In Fig 6.4, there are 3 sub-workloads, W_i, W_{i+1} and W_{i+2}. A test period is dynamically started when the reuse distance of $rd(b_k)$ is not in the range of the current window. However, the period is terminated at t_{k+p} because the next request is hit in W. In the second test period, n consecutive requests are detected to have larger reuse distance than the current minimal reuse distance given a test period of $n$having an out-of-range reuse distance are detected; where n is a threshold for the period. Therefore, the algorithm takes the minimal one, $rd(b_{l+q})$, as a new minimal distance, d_{i+1}. The index value of the block b_{l+q} is set to q. The last switching occurs when a reuse distance of a request, $rd(b_m)$, is smaller than the current minimal distance, i.e., $rd(b_m) < d_{i+1}$. In that case, it is immediately applied without a test period; the $rd(b_m)$, as a new minimal distance, d_{i+2}, and the index value of the block b_m is set to q. Please refer to the lines including the `GET_INDEX` procedure in Algorithm 4.

If the q value is changed by the speculation process, the window, W, is moved to the q position; B_1 is allocated with HB[q:$q + window\_size$] and T_1 is flushed. Optionally, the data blocks referenced by T_1 can be maintained with a separated 'Invalid Block' list for performance enhancement.



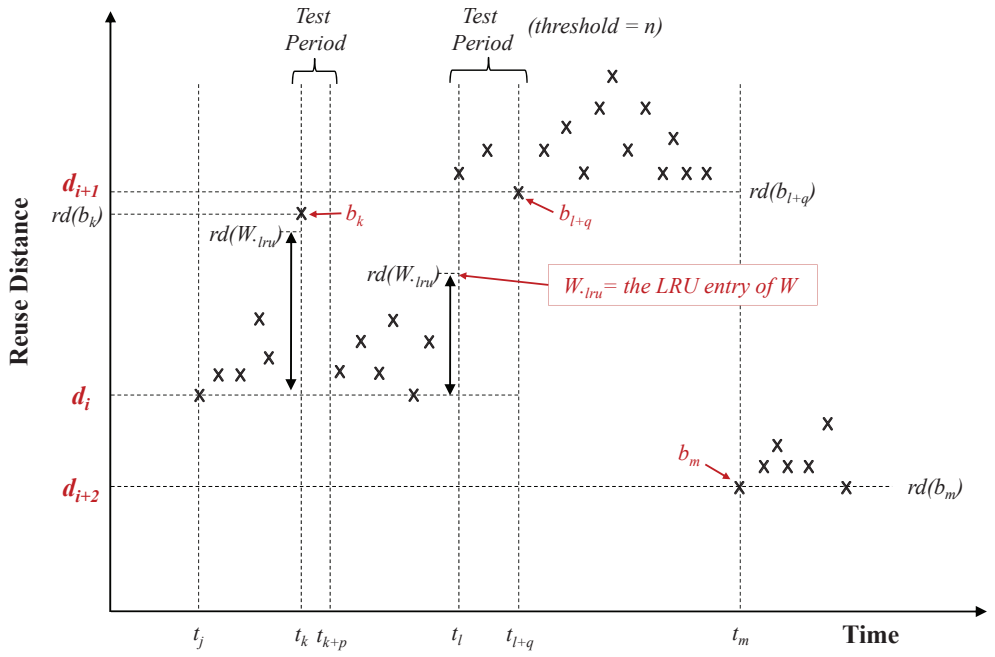


Fig. 6.4: Window control in the RARC algorithm: there are 3 sub-workloads, W_i, W_{i+1} and W_{i+2}. d_i, d_{i+1}, and d_{i+2} denote the minimal reuse distance of each sub-workload; the parameter q on each stage is set to the index value of the blocks, b_k, b_{l+1}, and b_m in HB.

Algorithm 4: ADJUST_WINDOW(b_t, d, q)

Input: The request stream $b_1, b_2, \ldots, b_t, \ldots$
Init. : Set $p \leftarrow 0$, $q \leftarrow 0$, $d \leftarrow 0$ on the cache initialization time.
`/* Estimate the minimal distance of the current workload. */`
$q\_old \leftarrow q$;
$window\_size \leftarrow |T_1| + |B_1|$;
$rd \leftarrow$ `get_rd` (b_t);
if $rd \leq d$ **then**
  $d \leftarrow rd$;
  $q \leftarrow$ `GET_INDEX` (HB, b_t)
else
  if ($not\ b_t$ in T_1) and ($not\ b_t$ in B_1) **then**
    if $test\_period = False$ **then**
      $test\_period \leftarrow True$;
      $d_{tmp} \leftarrow rd$;
      $q_{tmp} \leftarrow$ `GET_INDEX` (HB, b_t);
    else
      if $rd < d_tmp$ **then**
        $d_{tmp} \leftarrow rd$;
        $q_{tmp} \leftarrow$ `GET_INDEX` (HB, b_t)
      $test\_count \leftarrow test\_count + 1$;
  else
    $test\_period \leftarrow False$;
    $test\_count \leftarrow 0$;
  if (($test\_period = True$) and
    ($test\_count = test\_threshold$)) **then**
    $d \leftarrow d_{tmp}$;
    $q \leftarrow q_{tmp}$;
    $test\_period \leftarrow False$;
    $test\_count \leftarrow 0$;
`/* if q is changed, then adjust the current window. */`
if $q\_old \neq q$ **then**
  $B_1 \leftarrow$ HB[q:q+window_size];
  $T_1 \leftarrow \phi$;
  $p \leftarrow 0$;

CHAPTER 7

Evaluation

In this section, we evaluate the proposed RARC algorithm by mathematically proving that the algorithm is always outperforms the original ARC when a precise minimal distance is given.

We also present experimental results for the proposed algorithm using trace-driven simulation; we implemented RARC as well as the ARC and MQ algorithms and ran simulations with the following workload traces; two NFS Workload traces collected by Harvard University [43], two Disk I/O traces of desktop PCs collected by Brigham Young University [36], and three disk I/O traces of web and research servers collected by the Microsoft Research Cambridge Lab [28], [29]; Table 7.1 summarizes the characteristics of these workloads.

7.1 Mathematical evaluation

This section provides a mathematical evaluation of the RARC algorithm.

Theorem 1

Let $hr_{CA(c)}$ be the hit ratio of the cache algorithm, CA, where c is the size of the cache.

$$hr_{RARC(c)} \geq hr_{ARC(c)}, \text{ for any same workload } W \tag{7.1}$$

For the workload W, d_{min}, d_{max} is given as the *minimal* and the *maximal distance*, respectively.

In order to formulate hit ratio functions for some cache algorithms, we define the distribution of block requests, H, as follows. In the following definitions, b_t is a block request of a workload W and its request time is t. The $rd(b_t)$ is a function that returns a reuse distance of b_t.

$$h(x) = \{b_t \mid rd(b_t) = x,\ b_t \in W_i\} \tag{7.2}$$

$$H = h(d_{min}) \cup h(d_{min}+1) \cdots \cup h(x) \cdots \cup h(d_{max}) \tag{7.3}$$

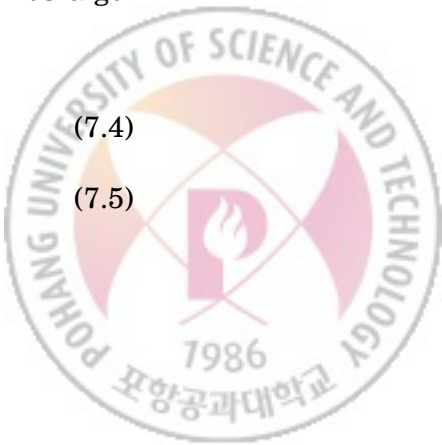
The $h(x)$ is a function that selects a set of blocks which have a same reuse distance, x, from all accesses in the workload W. Then, the distribution, H, can be expressed as a union of $h(x)$, where x is ranging from the *minimal* to the *maximal distance* of a workload.

Lemma 1

Using the $h(x)$ function, we can define the hit ratio function of the LRU algorithm as follows.

$$hr_{LRU(c)} \approx \frac{\int_{d_{min}}^{c} |h(x)|dx}{|W_i|} \quad (\text{if } 0 \leq d_{min} < c) \tag{7.4}$$

$$= 0 \quad (\text{if } \ d_{min} \geq c) \tag{7.5}$$



Lemma 2

The hit ratio function of ARC and RARC can be formulated from the relation of each algorithm. First of all, $hr_{ARC(c)}$ is defined as follows.

$$hr_{ARC(c)} = hr_{LRU(c)} + \alpha, \quad where\, \alpha \geq 0 \tag{7.6}$$

In the previous research [25, 26], it is experimentally proved that the ARC algorithm always outperforms the LRU algorithm for various workloads.

Lemma 3

The hit ratio function of RARC can also be formulated from the relation of the ARC algorithm. From the intuition of Fig 5.4, we can simply infer that the hit ratio of $RARC(c)$ is equal to or smaller than the hit ratio of $ARC(c + d_{min})$.

$$\begin{aligned} hr_{RARC(c)} &= hr_{ARC(c+d_{min})} - \beta, \\ &where\, 0 \leq \beta \leq hr_{ARC(c+d_{min})} \end{aligned} \tag{7.7}$$

Proof of the theorem

We prove the theorem by induction using the lemmas 1, 2, and 3.To do this, we consider the following two cases.

- Case I : $0 \leq d_{min} < c$
  - **Basis step:**
    when $d_{min} = 0$, we have $hr_{ARC(c)} = hr_{RARC(c)}$, therefore the theorem is true.
  - **Inductive step:**
    Let $d_{min} = n$, and we assume that $hr_{ARC(c)} \leq hr_{RARC(c)}$. We can get

equations (7.8) and (7.9) from equations (7.4), (7.6) and (7.7).

$$hr_{ARC(c)} = \frac{\int_n^c |h(x)|dx}{|W_i|} + \alpha_c \tag{7.8}$$

$$hr_{RARC(c)} = \frac{\int_n^{c+n} |h(x)|dx}{|W_i|} + \alpha_{c+n} - \beta_{c+n} \tag{7.9}$$

By the induction hypothesis,

$$\frac{\int_n^c |h(x)|dx}{|W_i|} + \alpha_c \leq \frac{\int_n^{c+n} |h(x)|dx}{|W_i|} + \alpha_{c+n} - \beta_{c+n} \tag{7.10}$$

$$\begin{aligned} \frac{\int_n^c |h(x)|dx}{|W_i|} + \alpha_c \leq \frac{\int_n^c |h(x)|dx + \int_c^{c+n} |h(x)|dx}{|W_i|} \\ + \alpha_{c+n} - \beta_{c+n} \end{aligned} \tag{7.11}$$

$$\therefore \alpha_c \leq \frac{\int_c^{c+n} |h(x)|dx}{|W_i|} + \alpha_{c+n} - \beta_{c+n} \tag{7.12}$$

When $d_{min} = n + 1$, we have the following formula by (7.10)

$$\begin{aligned} \frac{\int_{n+1}^c |h(x)|dx}{|W_i|} + \alpha_c \leq \frac{\int_{n+1}^{c+n+1} |h(x)|dx}{|W_i|} \\ + \alpha_{c+n+1} - \beta_{c+n+1} \end{aligned}$$

If we assume that the α_c in inequation (7.12) has the maximum value, then we can formulate it as follows.

$$\begin{aligned} \frac{\int_{n+1}^c |h(x)|dx}{|W_i|} + \frac{\int_c^{c+n} |h(x)|dx}{|W_i|} + \alpha_{c+n} - \beta_{c+n} \\ \leq \frac{\int_{n+1}^{c+n+1} |h(x)|dx}{|W_i|} + \alpha_{c+n+1} - \beta_{c+n+1} \end{aligned}$$

$$\frac{\int_{n+1}^{c+n} |h(x)|dx}{|W_i|} + \alpha_{c+n} - \beta_{c+n} \leq \frac{\int_{n+1}^{c+n+1} |h(x)|dx}{|W_i|} + \alpha_{c+n+1} - \beta_{c+n+1} \quad (7.13)$$

Adding $\frac{\int_{n}^{n+1} |h(x)|dx}{|W_i|}$ to both sides will allow us to substitute the left and the right sides with $hr_{RARC(c)}$ and $hr_{RARC(c+1)}$, respectively, using equation (7.9).

$$hr_{RARC(c)} \leq hr_{RARC(c+1)}$$

$\therefore hr_{ARC(c)} \leq hr_{RARC(c)}$ for every $d_{min} < c$

- Case II : $d_{min} \geq c$

  If $d_{min} \geq c$, then $|T_2|$ of ARC will be 0 because of no hit in T_1 or B_1.

  $$\therefore hr_{ARC(c)} = 0$$

  $$hr_{RARC(c)} = hr_{ARC(d_{min}+c)} - \beta \geq 0 \quad \text{by (7.7)}$$

  Therefore, $hr_{ARC(c)} \leq hr_{RARC(c)}$ for every $d_{min} \geq c$.

7.2 Simulation results

Two NFS server and four disk I/O traces are used for simulations which are summarized in Table 7.1. For the RARC simulation, the threshold for test period is set to 3; this is determined experimentally and means the sub-workload is switched when 3 consecutive out-of-range hits on the history buffer occurred. We will explain the test threshold later in this section.

7.2.1 Cache hit ratio

For a performance comparison, we implement ARC and MQ. For the MQ implementation, eight queues are used and the history buffer Q_{out} is set to be four times the number of blocks in the cache. The *lifeTime* parameter is set to the *peak distance* of each workload. All these parameters of MQ have been empirically determined by their studies [48, 49].

The simulation results for NFS and disk I/O workloads are shown in Fig 7.1∼7.6. Fig 7.1-(a) and Fig 7.2-(a) present workload patterns of the Deasna2 and the Lair62b workload, respectively. The cache entry size was 8KB and we ran simulations for each replacement algorithm. For those two workloads, RARC greatly outperformed ARC and MQ when its cache size was equal to or smaller than the *peak distance* of each workload. The performance gap between RARC and other two algorithms was maximized when the cache size was set to 4K entries, i.e., 32MB was used; the hit ratio of RARC was improved by up to 10 times compared to ARC and MQ; for the Deasna2 workload, the hit ratio of RARC is about 0.22, but those of the ARC and MQ are lower than 0.02. In the case of the Lair62b, it was more than twice as high as other algorithms.

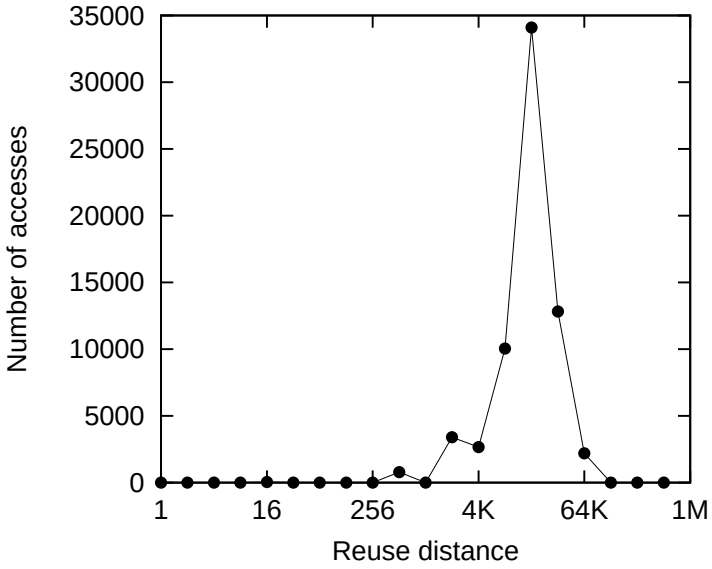
RARC also outperformed other algorithms for all Disk I/O cases when its cache size was smaller than the *peak distance* of each workload distribution. The workload Dev1 and User1 are disk I/O traces for a developer and a user

desktop, respectively. Therefore, their *peak distance* is relatively smaller than other traces. The Web0, however, has the largest *peak distance* among six workload traces. As the size of RARC increases, the RARC algorithm works identically to the original ARC algorithm from a certain point where the cache size of RARC is larger than the *peak distance* of each workload; it depends on the cache size as well as the workload pattern. The reason is that the window does not need to be controlled when its size is sufficiently larger than the reuse distance variation. Each simulation result showed this characteristic.

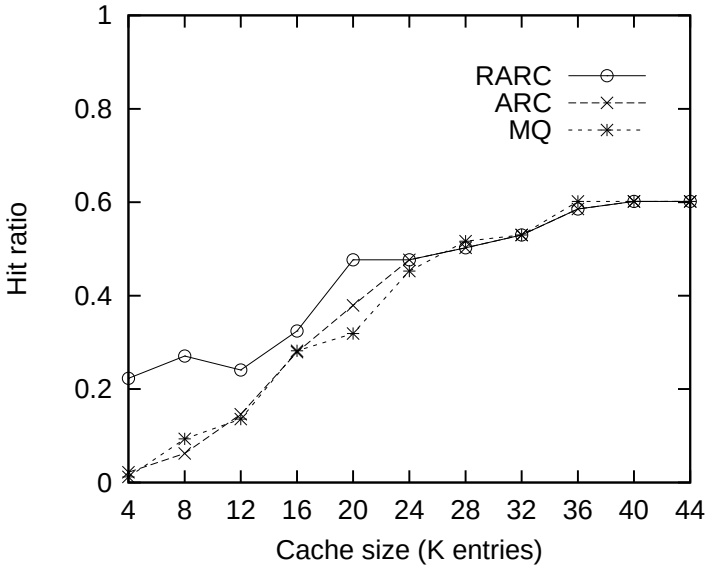
Performance enhancement of RARC for the workloads User1, Web0 and Usr0 is relatively smaller than other cases. Refer to Fig 7.4-(b), 7.5-(b), and 7.6-(b). We infer that it may be affected by its workload pattern like randomness and the distribution of file size; for example, the average file size of Web servers may be quite smaller than others and also SQL servers commonly makes random and small reads. The speculation accuracy for the minimal distance may degrade in such cases.

Workload	Description
Deasna2 (NFS)	A trace of a general workload from the division of engineering and applied sciences: a mix of email and research workloads. 1 hour I/O trace was used for our simulation. (Harvard University, 2003)
Lair62b (NFS)	A trace of a resarch workload from a university computer science department: a mix of email and research workload. 1 hour I/O trace was used for our simulation. (Harvard University, 2003)
Dev1 (Disk I/O)	A trace of a developer desktop and application usage: collected over 15 consecutive days. Contains text editor, compiler, IDE, browser, email, and desktop environment usage (Brighham Young University, 2001)
User1 (Disk I/O)	A trace of a user desktop and application usage: collected over 2.5 consecutive days. Contains browser, email, messaging, music listening, and desktop environment usage (Brighham Young University, 2001)
Web0 (Disk I/O)	A trace of a Web/SQL server collected over 1 week (Microsoft Research Cambridge Lab., 2007)
Usr0 (Disk I/O)	A trace of user home directories: collected over 1 week (Microsoft Research Cambridge Lab., 2007)

Table 7.1 Workload Traces

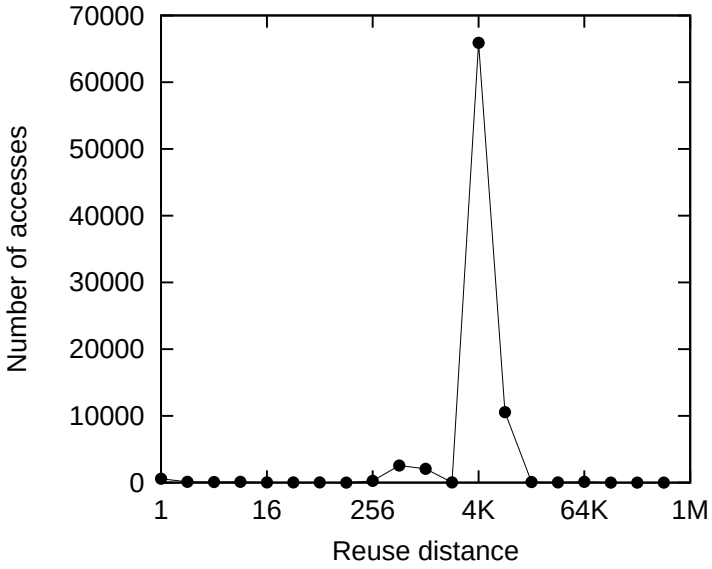


(a) Deasna2: Workload pattern

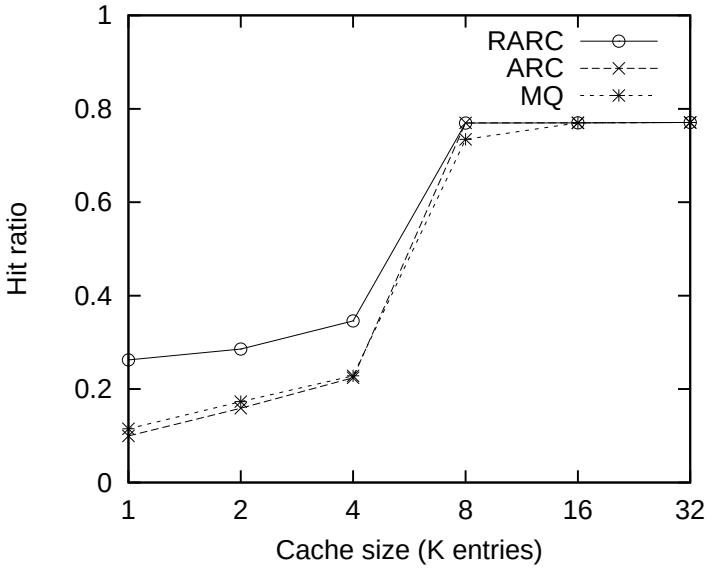


(b) Deasna2: Cache Hit Ratio

Fig. 7.1: Simulation Results for NFS Server Traces (1-hour I/O traces from Deasna2) : the cache entry size is 8KB.

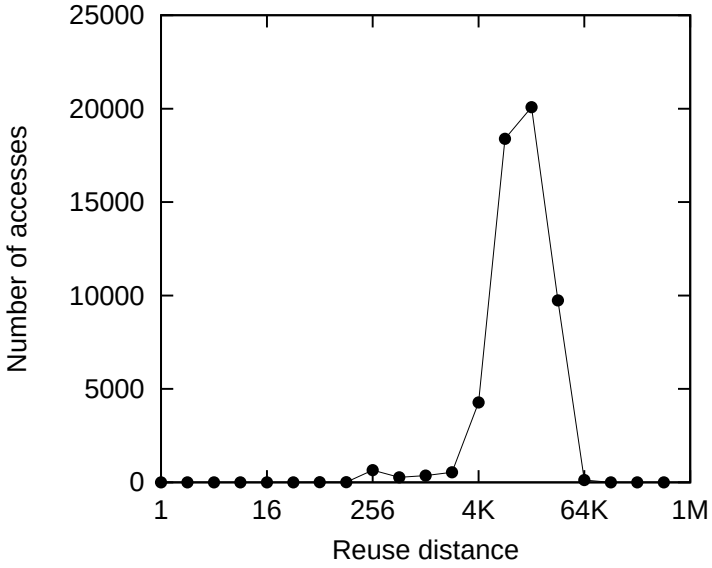


(a) Lair62b: Workload pattern

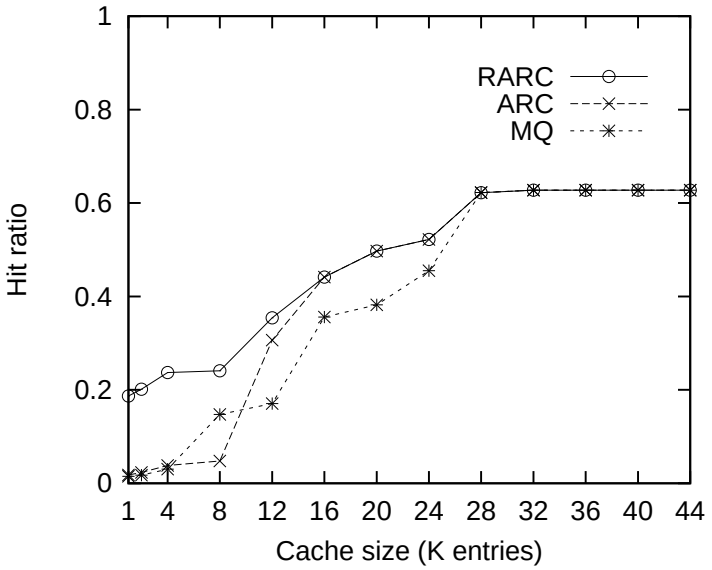


(b) Lair62b: Cache Hit Ratio

Fig. 7.2: Simulation Results for NFS Server Traces (1-hour I/O traces from Lair62b) : the cache entry size is 8KB.

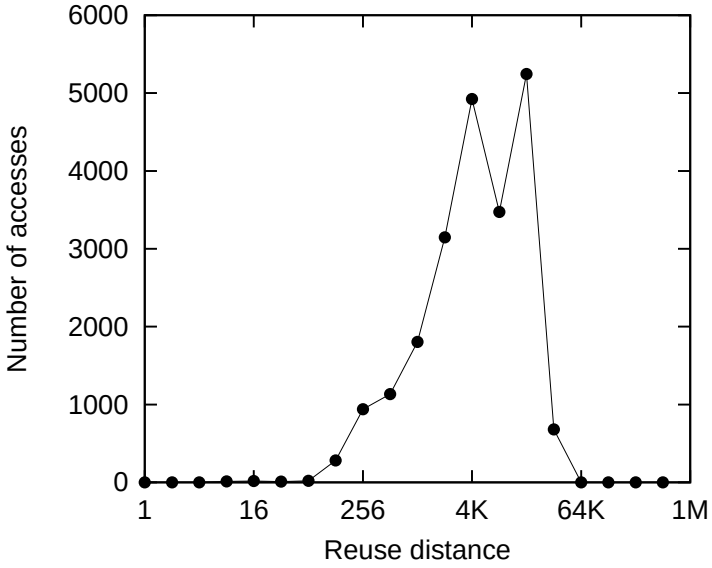


(a) Dev1: Workload pattern

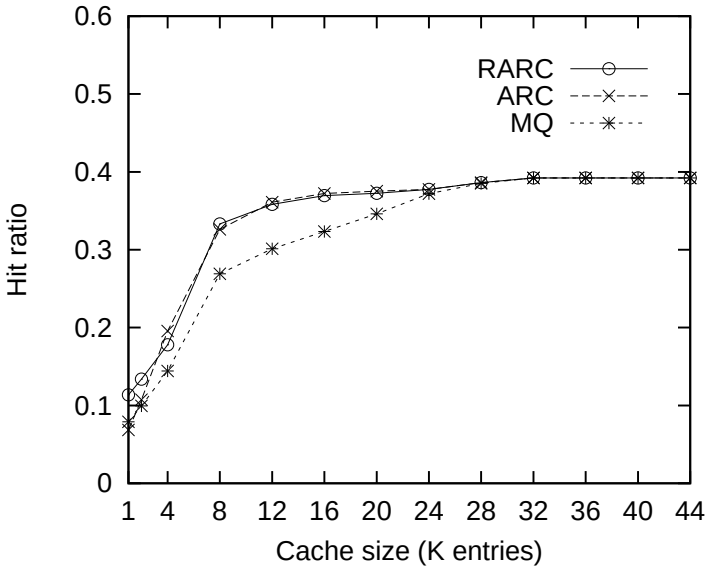


(b) Dev1: Cache Hit Ratio

Fig. 7.3: Simulation Results for BYU traces (Dev1) : the cache entry size is 4KB.

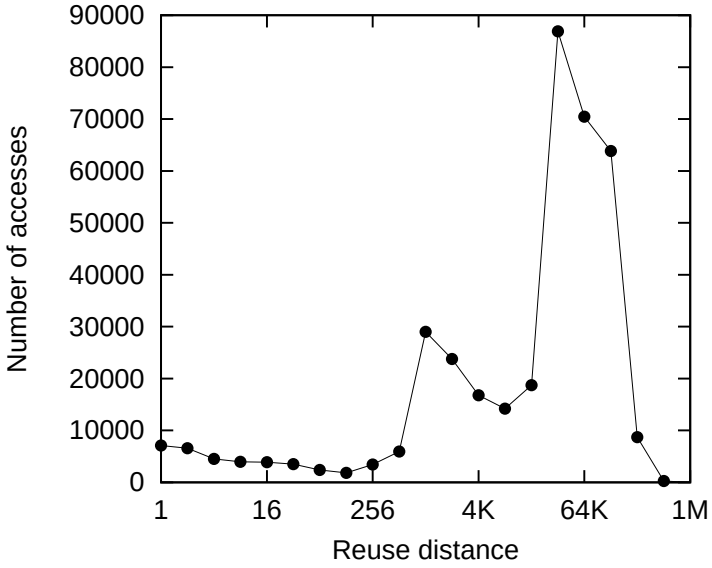


(a) User1: Workload pattern

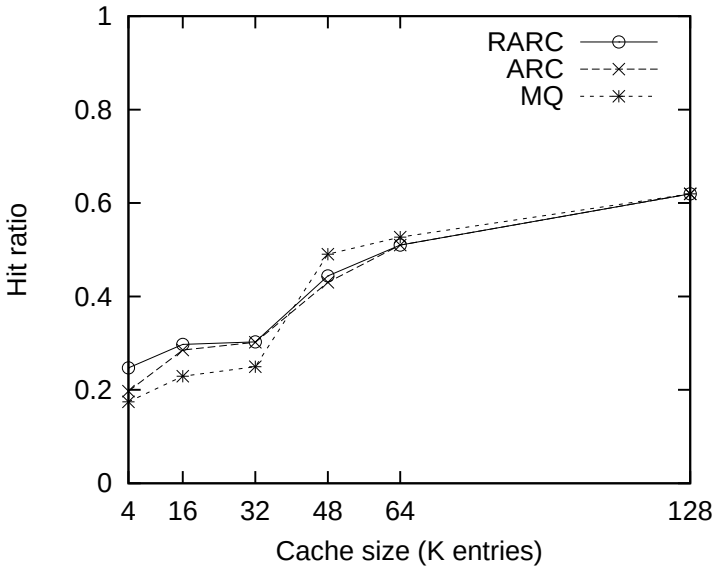


(b) User1: Cache Hit Ratio

Fig. 7.4: Simulation Results for BYU traces (User1) : the cache entry size is 4KB.

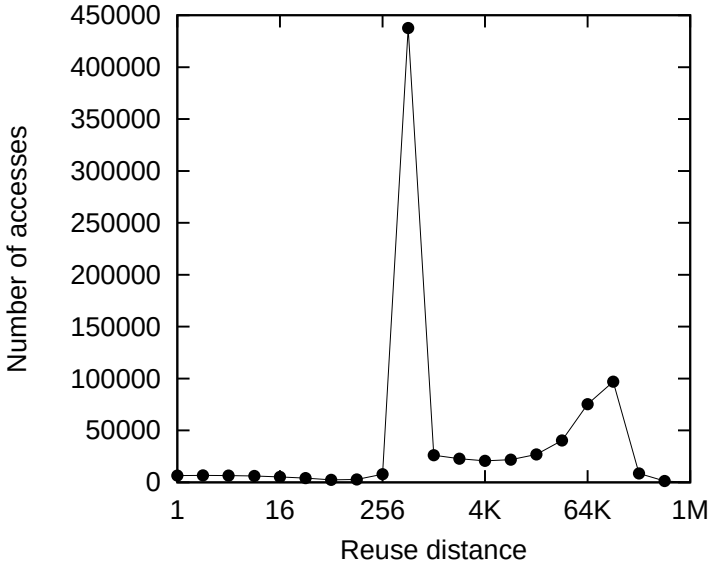


(a) Web0: Workload pattern

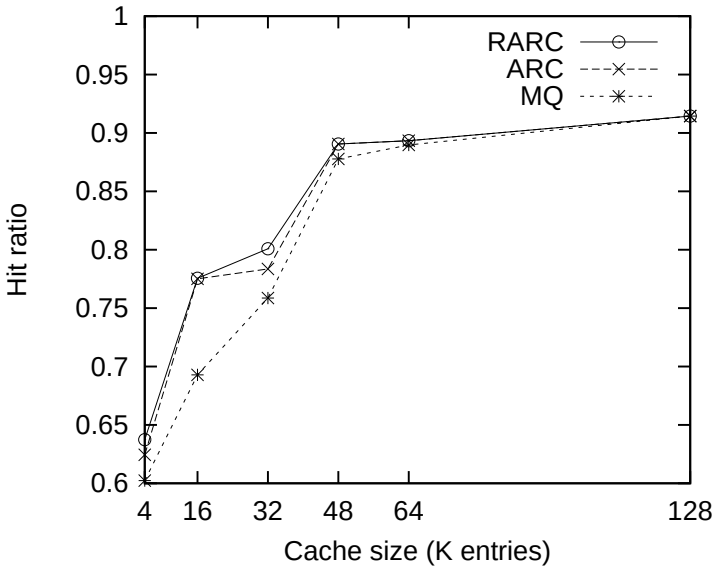


(b) Web0: Cache Hit Ratio

Fig. 7.5: Simulation Results for MSR-C traces (Web0) : the cache entry size is 4KB.



(a) Usr0: Workload pattern



(b) Usr0: Cache Hit Ratio

Fig. 7.6: Simulation Results for MSR-C traces (Usr0) : the cache entry size is 4KB.

7.2.2 Exclusiveness

For better performance in multi-level caches, we need to maintain the cache of each level exclusively. Both of RARC and X-RAY monitor read I/O requests in order to find out what blocks are maintained in the upper-level cache. RARC keeps track of the reuse distance of block accesses while X-RAY tries to collect the upper-level file system semantic information by their own heuristics. The file system information exploited by X-RAY includes a file access time and which data blocks belong to the file. This is done by monitoring inode writes issued periodically.

Because RARC only monitors read requests with test periods, the speculation accuracy may be lower than X-RAY. In addition, some block accesses may lose the chance of cache hits during the test period. Despite these drawbacks, our algorithm does not require such excessive I/O monitoring cost in X-RAY.

In order to compare the cache hit ratio of X-RAY and RARC, the global LRU algorithm [48] is applied for simulations instead of X-RAY because of its implementation and deployment problem. We assume that the read cache size in the first-level cache is not changed; it is fixed to 8K entries, and the 253.perlbmk-diffmail workload trace described in section 5.3 is used as input stream for the simulations. Fig 7.7 indicates that the performance of RARC algorithm is slightly lower than but comparable to that of the global LRU algorithm.

Speculation accuracy is an important factor which affects cache performance. In RARC, it is controlled by the threshold of the test period. In order to determine the optimal value, we ran simulations for six workloads with various size configurations. RARC showed the best performance when the threshold for the test period was set to 3 for most cases. According to the X-RAY paper [3], randomness and file size distribution of I/O workload affects speculation accuracy; they demonstrated this characteristics by applying their algorithm to a real storage system and by analyzing it with synthetic I/O benchmarks.

Fig 7.8 demonstrates how the window control method keeps track of minimal distance changes. For the simulation, 15 min I/O trace of Deasna2 was used. These two examples show that the method correctly determines the minimal distance of each stage by considering their window size. In the example (a), the minimal distance did not change until about 4,000 cache hits occurred because reuse distances did not exceed the given cache size, i.e., the window size. In contrast, in (b), it changed on the first stage; by the same token, similar cases also occurred at the time when about 6,000 and 7,000 cache hits occurred.

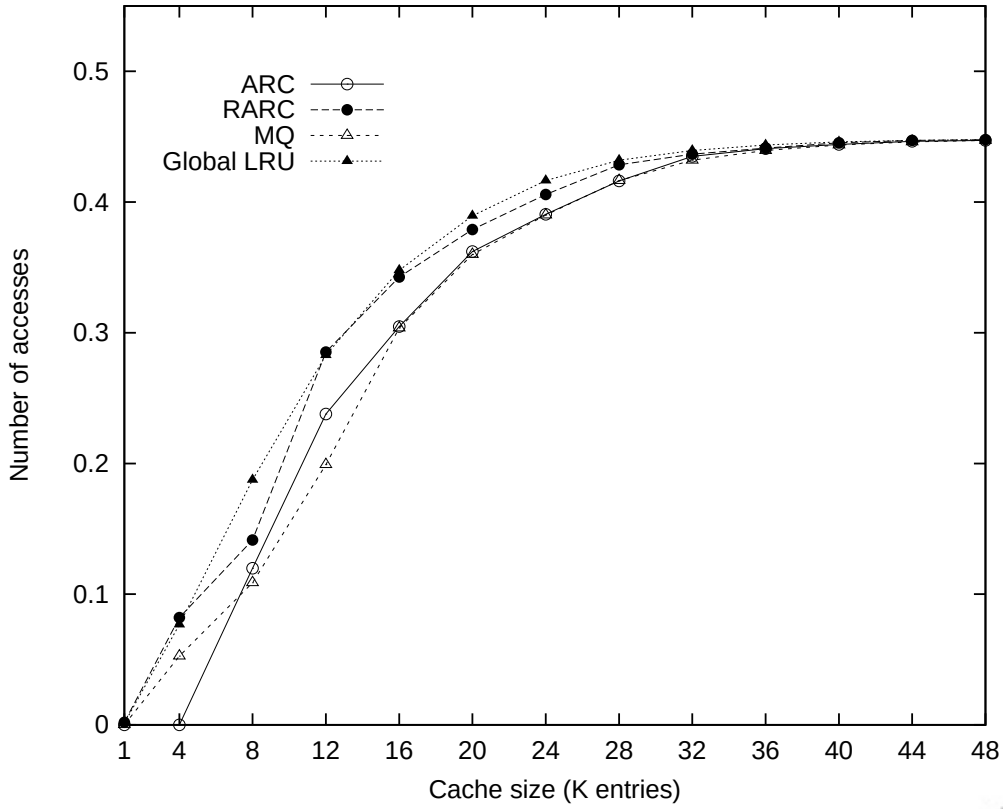
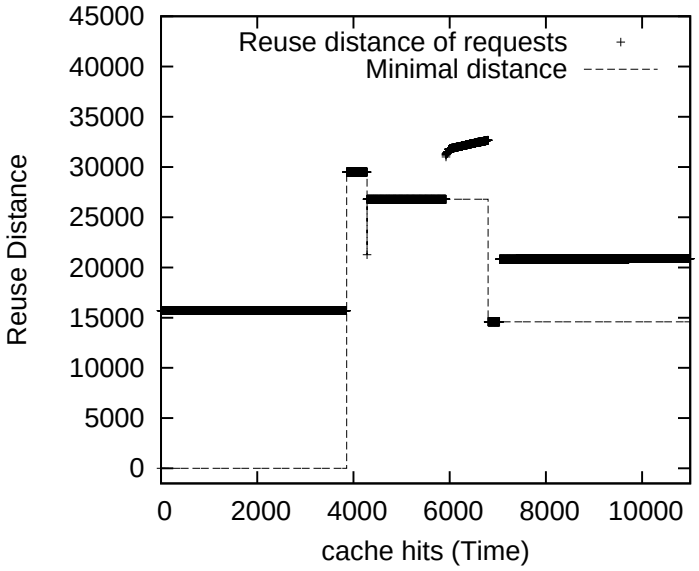
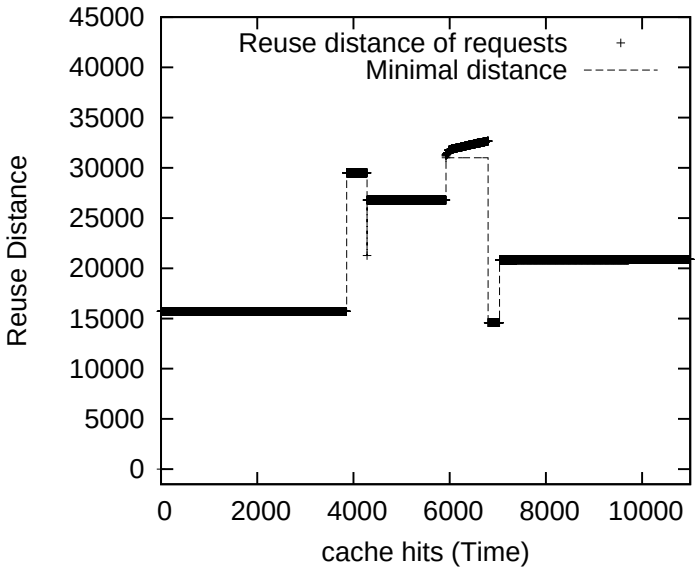


Fig. 7.7 Comparison of ARC, MQ, Global LRU and RARC

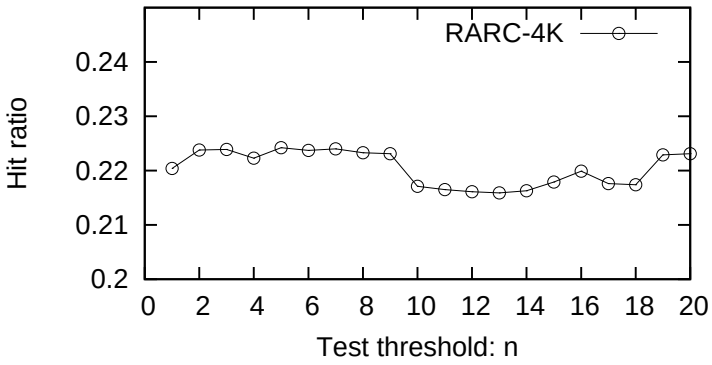


(a) The cache size: 16K entries

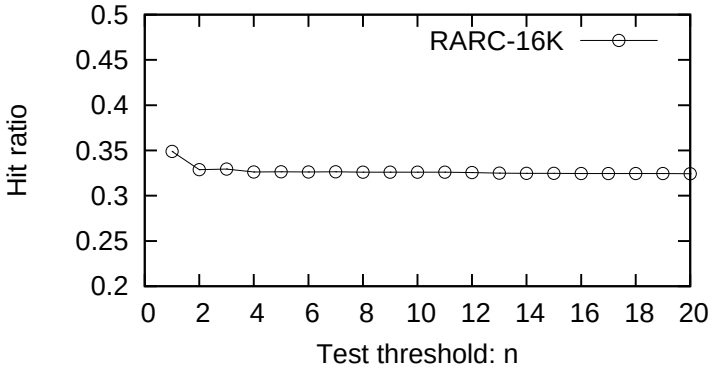


(b) The cache size: 4K entries

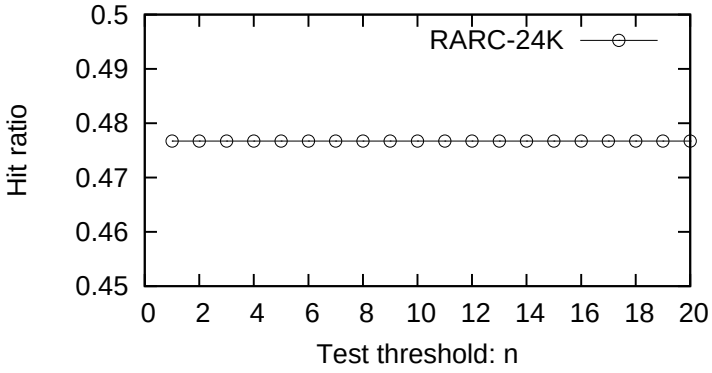
Fig. 7.8 Reuse distance of requests and Minimal distance speculation: 15min I/O trace of Deasna2 was used for this simulation.



(a) Deasna2: 4K entries - the cache entry size is 8KB.

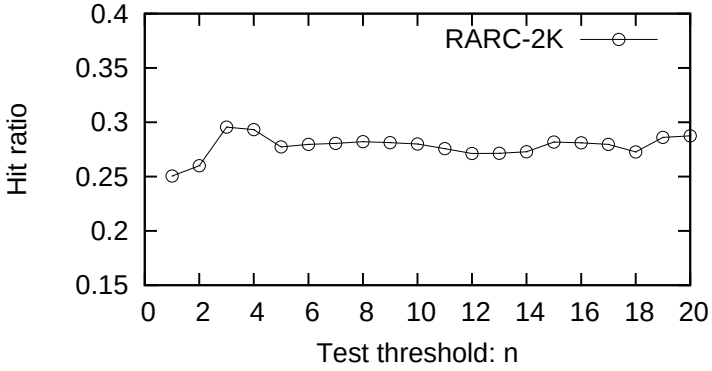


(b) Deasna2: 16K entries

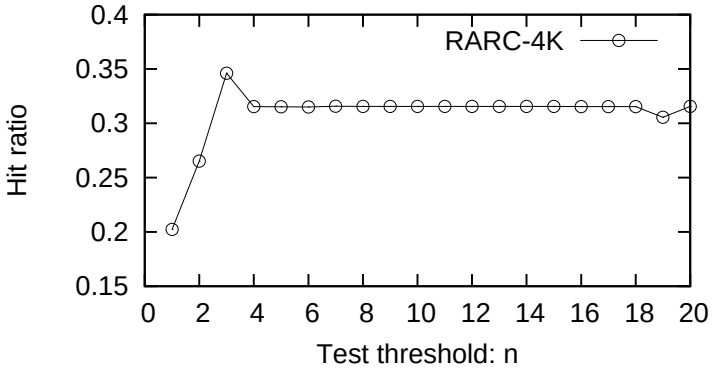


(c) Deasna2: 24K entries

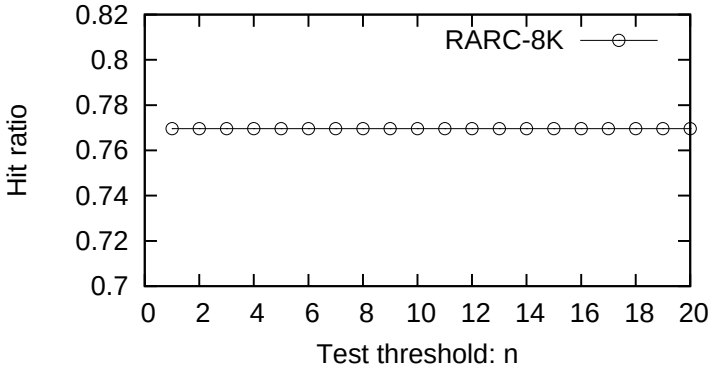
Fig. 7.9: Simulation Results for the Test Period (Deasna2)



(a) Lair62b: 2K entries - the cache entry size is 8KB.

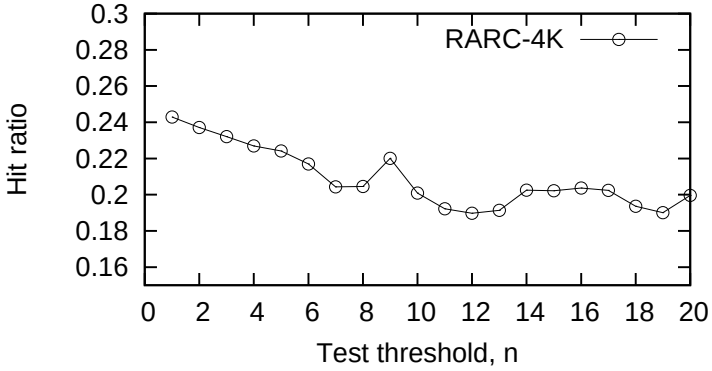


(b) Lair62b: 4K entries

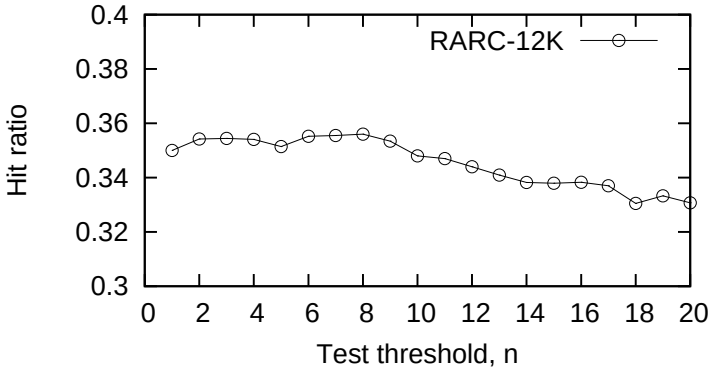


(c) Lair62b: 8K entries

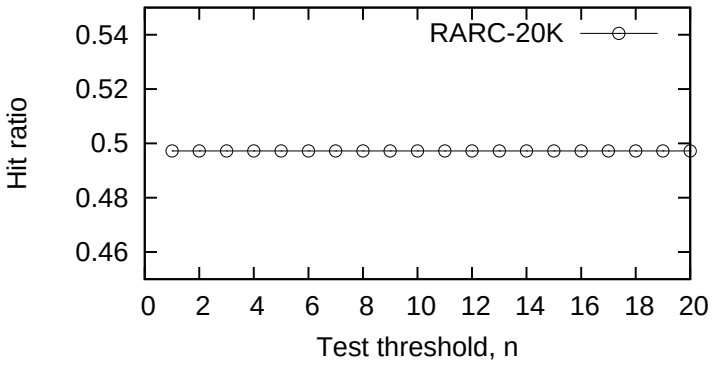
Fig. 7.10: Simulation Results for the Test Period (Lair62b)



(a) Dev1: 4K entries - the cache entry size is 4KB.

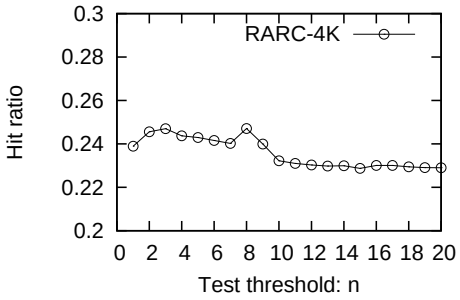


(b) Dev1: 12K entries

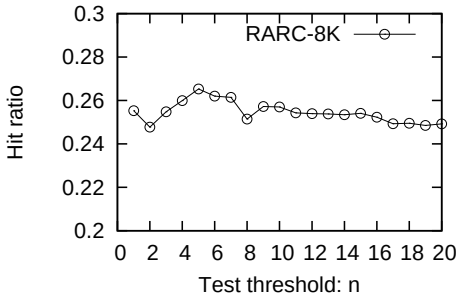


(c) Dev1: 20K entries

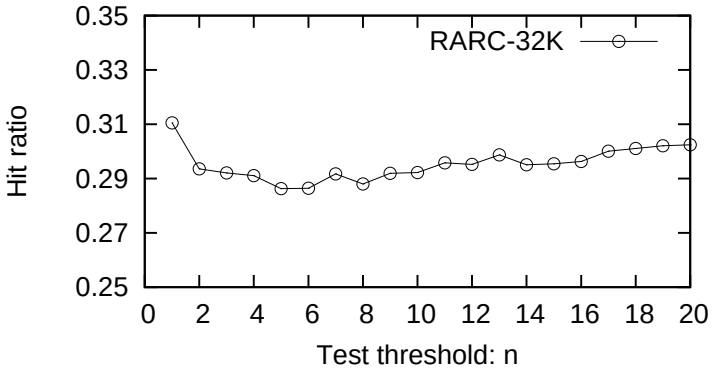
Fig. 7.11: Simulation Results for the Test Period (Dev1)



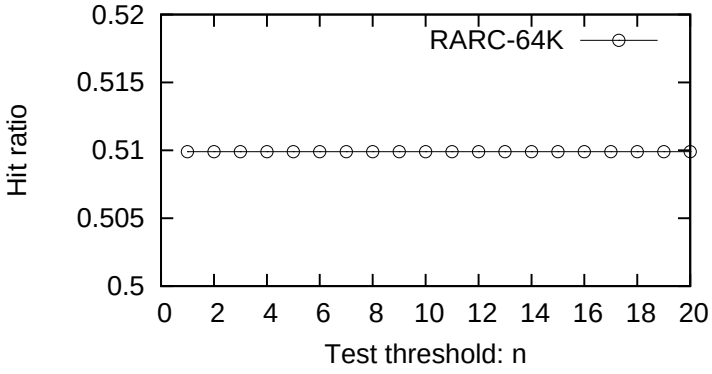
(a) Web0: 4K entries



(b) Web0: 8K entries



(c) Web0: 32K entries



(d) Web0: 64K entries - the cache entry size is 4KB

Fig. 7.12: Simulation Results for the Test Period (Web0)

CHAPTER 8

Conclusion and Future work

The Adaptive Replace Cache (ARC) dynamically balances recency and frequency using two LRU queues in response to changing access patterns. Due to its simplicity, it has low computational overhead compared to other algorithms while performing well across varied workloads.

However, on the second-level, ARC cannot perform efficiently when the cache size is equal to or smaller than the first-level one. In that case, the reuse distance of most I/O accesses on second-level will exceed its cache size so that the recency queue of ARC will not contribute to the cache hits. Furthermore, cache hits near the LRU position of the recency queue cause shrinking of the frequency queue as well. Thus, the block frequency also cannot be captured well by the buffer cache.

In this thesis, we have proposed the RARC algorithm where the reuse dis-

tance pattern was considered by designing the recency queue as a sliding window over a long history buffer for I/O requests. In the proposed algorithm, only blocks within the range of the window are retained by the cache using background I/O, and the window position is dynamically determined by speculating the minimal reuse distance of a workload. By using the sliding window technique, block data can be fetched to the buffer cache before each block re-referenced which ensures cache hits always occur on the front area of the recency queue. The efficiency of the frequency queue can be boosted with the increased queue size compared to the case of using the original ARC.

Mathematical evaluation and simulations have shown that the ARC algorithm outperforms the origial ARC and the Multi-Queue(MQ) algorithm. The simulation results also demonstrated that the proposed method efficiently provides a certain degree of exclusive caching like X-RAY without any semantic information of upper-level file systems; Table 8.1 summarizes the pros and cons of each replacement algorithm.

In future work, in-depth evaluation for the test period should be performed because RARC may cause uneven performance by making oscillation for the window control. The performance effects for speculation accuracy and background I/O must be evaluated with the application of the proposed algorithm to real storage systems.

	ARC	MQ	DEMOTE	X-RAY	RARC
Fetures	Adaptive caching in response to changing access pattern(recency and frequency)	Prioritization of block accesses with frequency (with limited consideration for reuse distance pattern)	Exclusive caching (with I/O interface modification)	Exclusive caching (with file system information tracking)	Adaptive caching in response to changing access pattern(recency and frequency), Reuse-distance aware block replacement
Pros	. simple to implement . constant time overhead per request . low operating cost compared to other policies	. constant time overhead per request . outperforming most replacement policies for second-level buffer caches	. best performance (perfectly exclusive)	. outperforming MQ and ARC (a high degree of exclusivity provided)	. simple to implement . constant time overhead per request . low operating cost compared to other policies . outperforming MQ and ARC (a certain degree of exclusivity provided)
Cons	. no consideration for the second-level workload characteristic (caching efficiency becomes poor when the cache size is smaller than minimal or peak distance)	. relatively high operating cost due to expiration time scanning and list management . limited consideration for reuse distance distribution . less efficient than ARC for low-frequency workloads	. hard to deploy directly to off-the-shelf storage systems because of the interface modification	. excessive I/O monitoring overhead . massive background block reads required (when drastic cache size changes are occurring on the upper-level cache during a certain time interval)	. uneven performance due to mistaken speculation for the minimal distance when drastic cache size changes are occurring on the upper-level cache

Table 8.1: Comparison of ARC, MQ, DEMOTE, X-RAY and RARC

APPENDIX A

The ARC algorithm

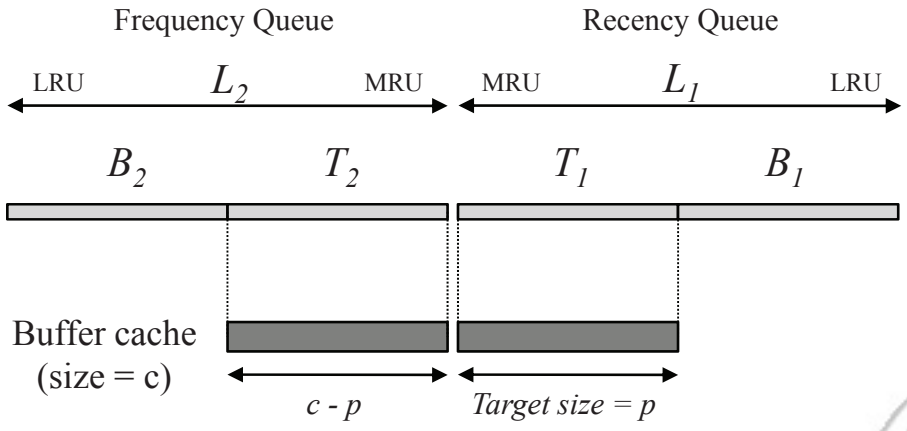


Fig. A.1 Data Structures of the Adaptive Replacement Cache (ARC) algorithm

Algorithm 5: ARC, $cache\_size = c$

Input: The request stream $b_1, b_2, \ldots, b_t, \ldots$
Init. : Set $p \leftarrow 0$ and set the LRU lists T_1, B_1, T_2, and B_2 to empty

For every $t \geq 1$ and any b_t, one and only one of the following four cases must occur.

if b_t is in T_1 or T_2. **then**
  `// A cache hit has occurred.`
  Move b_t to MRU position in T_2.

else if b_t is in B_1. **then**
  `// A cache miss has occurred.`
  Update $p \leftarrow$ `min` $(p + \delta_1, c)$;
    where δ_1 = {1, (if $|B_1| \geq |B_2|$) or $|B_2|/|B_1|$, (otherwise) }
  `REPLACE` (b_t, p);
  Move b_t from B_1 to the MRU position in $T2$.
  Fetch b_t to the cache.

else if b_t is in B_2. **then**
  `// A cache miss has occurred.`
  Update $p \leftarrow$ `max` $(p - \delta_2, 0)$;
    where δ_2 = {1, (if $|B_2| \geq |B_1|$) or $|B_1|/|B_2|$, (otherwise) }
  `REPLACE` (b_t, p);
  Move b_t from B_2 to the MRU position in $T2$.
  Fetch b_t to the cache.

else
  `// A cache miss has occurred.`
  `//` b_t `is not in` $T_1 \cup B_1 \cup T_2 \cup B_2$.
  `CHECH_SIZE_AND_REPLACE` (b_t, p);
  Fetch b_t to the cache and move it to the MRU position in T_1.

end

Algorithm 6: REPLACE (b_t, p)

if (($|T_1|$ is not empty) and
(($|T_1| > p$) or (b_t is in B_2 and $|T_1| = p$))) **then**
Delete the LRU page in T_1 (also remove it from the cache), and move it to the MRU position in B_1.
else
Delete the LRU page in T_2 (also remove it from the cache), and move it to the MRU position in B_2.

Algorithm 7: CHECK_SIZE_AND_REPLACE (b_t, p)

if $T_1 \cup B_1$ has exactly c pages. **then**
if $|T_1| < c$ **then**
Delete the LRU page in B_1.
REPLACE (b_t, p);
else
Here B_1 is empty. Delete the LRU page in T_1.
(also remove it from the cache)
else
// $L_1 = T_1 \cup B_1$ has less than c pages.
if $|T_1| + |T_2| + |B_1| + |B_2| \geq c$ **then**
if $|T_1| + |T_2| + |B_1| + |B_2| = 2c$ **then**
Delete the LRU page in B_2.
REPLACE (b_t, p);

Bibliography

[1] A. Aho, P. Denning, and J. Ullman. Principles of optimal page replacement. *Journal of the ACM (JACM)*, 18(1):80–93, 1971.

[2] S. Ames, N. Bobb, K. Greenan, O. Hofmann, M. Storer, C. Maltzahn, E. Miller, and S. Brandt. LiFS: An attribute-rich file system for storage class memories. In *Proceedings of the 23rd IEEE / 14th NASA Goddard Conference on Mass Storage Systems and Technologies*. Citeseer, 2006.

[3] L. Bairavasundaram, M. Sivathanu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 176. IEEE Computer Society, 2004.

[4] P. Chirita, S. Costache, W. Nejdl, and R. Paiu. Beagle++: Semantically enhanced searching and ranking on the desktop. *The Semantic Web: Research and Applications*, pages 348–362, 2006.

[5] D. Gifford, P. Jouvelot, M. Sheldon, et al. Semantic file systems. *ACM SIGOPS Operating Systems Review*, 25(5):16–25, 1991.

[6] Y. Goland, E. Whitehead, A. Faizi, and D. Jensen. HTTP extensions for distributed authoring-WEBDAV. 1999.

[7] C. Hess and R. Campbell. A context-aware data management system for ubiquitous computing applications. 2003.

[8] D. Hipp and D. KENNEDY. SQLite. *An Embeddable SQL Database Engine. http://www.sqlite.org*, pages 31–10, 2003.

[9] J. Howard and C.-M. U. I. T. Center. *An overview of the andrew file system*. Citeseer, 1988.

[10] G. Inc. Google desktop search. `http://desktop.google.com`.

[11] M. Jeronimo and J. Weast. *UPnP design by example*. Intel Press, 2003.

[12] S. Jiang, K. Davis, and X. Zhang. Coordinated multilevel buffer cache management with consistent access locality quantification. *IEEE Transactions on Computers*, pages 95–108, 2007.

[13] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 31–42. ACM, 2002.

[14] T. Johnson, D. Shasha, et al. 2Q: a low overhead high performance bu er management replacement algorithm. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 439–450. Citeseer, 1994.

[15] A. Karypidis and S. Lalis. Omnistore: A system for ubiquitous personal storage management. In *Pervasive Computing and Communications, 2006. PerCom 2006. Fourth Annual IEEE International Conference on*, pages 11–147. IEEE, 2006.

[16] K. Kline. *SQL in a Nutshell*. O'Reilly & Associates, Inc. Sebastopol, CA, USA, 2003.

[17] G. Klyne, J. Carroll, and B. McBride. Resource description framework (RDF): Concepts and abstract syntax. *Changes*, 2004.

[18] D. Lee, J. Choi, J. Kim, S. Noh, S. Min, Y. Cho, and C. Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE Transactions on Computers*, 50(12):1352–1361, 2001.

[19] W. Lee, Y. Hong, and C. Park. An agent framework for CE devices to support storage virtualization on device ensembles. In *Consumer Communications and Networking Conference, 2009. CCNC 2009. 6th IEEE*, pages 1–5. IEEE, 2009.

[20] W. Lee, S. Kim, and C. Park. PosCFS+: A Self-Managed File Service in Personal Area Network. *ETRI journal*, 29(3):281–291, 2007.

[21] W. Lee, S. Kim, J. Shin, and C. Park. Poscfs: An advanced file management technique for the wearable computing environment. *Embedded and Ubiquitous Computing*, pages 966–975, 2006.

[22] W. Lee and C. Park. An Event Notificaion Framework for PosCFS+ Distributed File Service in Personal Area Network. In *Proceedings of the 2nd International Conference on Next-Generation Computing*, volume 3, pages 22–31. Korea Institute of Next Generation Computing, 2007.

[23] J. Lehikoinen, A. Aaltonen, I. Salminen, and P. Huuskonen. *Personal content experience: managing digital life in the mobile age*. Wiley-Interscience, 2007.

[24] R. Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[25] N. Megiddo and D. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130. USENIX Association, 2003.

[26] N. Megiddo and D. Modha. Outperforming LRU with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2005.

[27] Microsoft. Microsoft's distributed file system (dfs). `http://technet.microsoft.com/en-us/library/cc757042(WS.10).aspx`.

[28] Microsoft. Msr cambridge traces. `ftp://ftp.research.microsoft.com/pub/austind/MSRC-io-traces`.

[29] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage (TOS)*, 4(3):1–23, 2008.

[30] E. Nightingale and J. Flinn. Energy-efficiency and storage flexibility in the blue file system. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation-Volume 6*, page 25. USENIX Association, 2004.

[31] W. Norcott and D. Capps. Iozone filesystem benchmark. *http://www.iozone.org*.

[32] E. O'neil, P. O'Neil, and G. Weikum. An optimality proof of the LRU-K page replacement algorithm. *Journal of the ACM (JACM)*, 46(1):92–112, 1999.

[33] M. Owens. Embedding an SQL database with SQLite. *Linux journal*, 2003(110):2, 2003.

[34] Y. Padioleau, B. Sigonneau, and O. Ridoux. Lisfs: a logical information system as a file system. In *Proceedings of the 28th international conference on Software engineering*, pages 803–806. ACM, 2006.

[35] D. Peek and J. Flinn. EnsemBlue: Integrating distributed storage and consumer electronics. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 219–232. USENIX Association, 2006.

[36] PEL. Byu trace distribution center: Disk i/o traces. `http://tds.cs.byu.edu/tds`.

[37] PEL. The spec 2000 benchmark: A perlbmk-diffmail trace collected by byu. `http://tds.cs.byu.edu/~samples/A2L000024.html`.

[38] H. Pennington. D-bus specification. `http://dbus.freedesktop.org/doc/dbus-specification.html`.

[39] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. *ACM SIGMETRICS Performance Evaluation Review*, 18(1):134–142, 1990.

[40] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on computers*, pages 447–459, 1990.

[41] B. Schilit and U. Sengupta. Device ensembles. *Computer*, pages 56–64, 2004.

[42] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, et al. NFS version 4 protocol. 2000.

[43] SNIA. Snia iotta repository: Network file system traces. `http://iotta.snia.org/traces/list/NFS`.

[44] M. Szeredi et al. FUSE: Filesystem in userspace. *Accessed on*, pages 06–16, 2008.

[45] E. Whitehead Jr and M. Wiggins. WebDAV: IEFT standard for collaborative authoring on the Web. *Internet Computing, IEEE*, 2(5):34–40, 2002.

[46] T. Wong, G. Ganger, and J. Wilkes. *My Cache Or Yours?: Making Storage More Exclusive*. School of Computer Science, Carnegie Mellon University, 2000.

[47] D. Zeuthen. Hal 0.5.10 specification. `http://www.marcuscom.com/hal-spec/hal-spec.html`.

[48] Y. Zhou, Z. Chen, and K. Li. Second-level buffer cache management. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):505–519, 2004.

[49] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 91–104, 2001.

요 약 문

본 논문에서는 데스크탑, 노트북, 스마트 폰 등의 개인용 단말 상 통합 데이터 관리를 지원하기 위한 파일 시스템 가상화 기법과, 제안된 파일 시스템의 성능 개선에 적용 가능한 2차 버퍼 캐시 관리 알고리즘을 소개한다.

1부에서는 PosCFS 자가 관리 분산 파일 시스템을 소개한다. 제안된 파일 시스템은 디바이스 앙상블(Device ensemble)로 정의되는 개인용 컴퓨터 단말 장치 간 협력 네트워크에서 파일 시스템 가상화를 지원한다. 개인용 휴대 장치가 다양화 됨에 따라, 사용자는 스마트 폰, 노트북, mp3 플레이어, 디지털 카메라 등 다양한 장치를 동시에 휴대하며, 이러한 환경에서 개인 데이터는 다양한 장치에 분산 저장되므로, 데이터 사용 및 관리가 어렵게 된다. 특히 장치들의 서로 다른 운영체제 및 특화된 네트워크 환경(Bluetooth, WiFi, UWB, USB 등)은 사용자의 복잡한 설정 과정을 요구하므로, 장치 간의 데이터 교환 및 동기화를 어렵게 만든다. 스토리지 기술의 발전과 더불어 이루어지고 있는 개인 데이터의 폭발적인 증가 또한 데이터 관리를 어렵게 만드는 요인 중 하나이다. 본 논문에서 제안하는 PosCFS 분산 파일 시스템은 이러한 문제를 해결한다. PosCFS는 개인 네트워크(Personal Area Network; PAN) 상에서 동적으로 모든 장치들의 물리적 스토리지를 하나의 가상 스토리지로 통합하여 사용자 및 응용프로그램에게 제공하며, 파일의 메타데이터를 기준으로 생성되는 가상 디렉토리를 스토리지 내의 파일 접근 주소로 제공함으로써, 데이터 접근성(data accessibility)을 개선한다. 플랫폼 중립성을 보장하기 위하여, 네트워크 기반의 스토리지 입출력은 WebDAV 프로토콜을 이용하여 정의되었으며, 파일 시스템 자동 구성과 가상 디렉토리 관리를 위한 확장 오퍼레이션은 UPnP 프로토콜을 이용하여 피어-투-피어 방식으로 구현하였다.

2부에서는 제안된 파일 시스템의 성능 개선에 활용 가능한 2차 버퍼 캐시 알고리즘을 설명한다. 네트워크 기반의 파일 및 스토리지 시스템에서는 일반적으로 클라이언트와 서버에서 별도의 버퍼 캐시가 관리되며, 파일 및 스토리지 서버 상 2차 버퍼 캐시의 입출력 패턴은 클라이언트 상의 1차 캐시와는 다른 특성을 보인다. 데이터 블록에 대한 재사용 거리(Reuse distance)는 특정 블록에 대한 읽기 요청이 처리된 이후, 동일한 블록에 대한 재 접근이 일어나기 전까지 발생한 입출력 접근의 개수로 정의 되며, 1차 캐시의 입출력 패턴과는 달리 긴 재사용 거리를 가지는 블록 접근의 개수가 짧은 거리를 가지는 블록 접근의 수보다 많은, 언덕 모양(hill-shaped)의 분포를 가진다 – 즉 상대적으로 약한 시간적 지역성(Temporal locality)을 가진다. 이는 상위 캐시에서 읽기 요청이 실패한 경우에만 하위 캐시로 읽기 요청이 전달되기 때문에 발생하는 현상으로써, Y. Zhou 등의 연구에 의해서 밝혀진 특성이다. ARC(Adaptive Replacement Cache) 알고리즘은 IBM의 N. Meggido에 의해 제안된 알고리즘으로써, 입출력 패턴의 최근성(Recency)과 반복성(Frequency)의 변화에 대응할 수 있도록 고안되었다. ARC 알고리즘은 두 개의 LRU(Least-Recently-Used) 큐를 이용하여 – 즉, 최근성 큐(Recency queue)와 반복성 큐(Frequency queue) – 입출력 접근이 높은 최근성을 보일 경우, 최근성 큐의 크기를 크게 유지하고, 반대로 높은 반복성을 보일 경우, 최근성 큐의 크기를 줄이고 반복성 큐를 크게 유지할 수 있도록 조절한다. ARC는 알고리즘의 간결함으로 인하여 낮은 관리 오버헤드를 가질 뿐 아니라, 입출력 워크로드 패턴의 변화에 대응함으로써, 대부분의 온-오프라인 캐시 알고리즘 보다 높은 성능을 보인다. 그러나 2차 캐시에 적용 시, 앞에서 소개한 워크로드 특성을 고려하지 못하므로 상대적으로 효율성이 떨어지게 되며, 특히 2차 캐시의 크기가 1차 캐시의 크기와 같거나 혹은 작게 유지되는 경우 급격히 효율이 떨어지게 된다. 그 이유는 2차 캐시로 전달되는 대부분의 입출력 접근의 재사용 거리가 2차 버퍼 캐시 크기보다 커지게 되기 때문이다. 추가적으로, 최근성 큐의 LRU 위치에서 발생하게 되는 빈번한 캐시 히트(Cache hit)는 반복성 큐의 크기를 작게 만들면서 워크로드의 반복성을 효율적으로 잡아 낼 수 없도록 만든다. 이러한 문제를 해결하기 위하여, 본 논문에서는 ARC

상 재사용 거리를 고려한 블록 관리 방법을 제안한다. 제안하는 알고리즘은 2차 캐시에서의 시간적 지역성 특성에 대응하기 위하여 입출력 로깅(logging)을 위한 충분히 큰 크기의 히스토리 버퍼(history buffer)를 구성하고, 최근성 큐를 히스토리 버퍼상의 슬라이딩 윈도우(sliding window)로 정의한다. 워크로드의 최소 재사용 거리(minimal reuse distance)는 알고리즘에 의해 지속적으로 추적되며, 추적된 값을 기준으로 윈도우의 위치가 결정된다. 마지막으로, 수학적 분석 및 시뮬레이션을 통한 성능 평가를 통하여 제안하는 알고리즘이 낮은 관리 오버헤드를 가지면서도, 기존의 2차 버퍼 캐시 알고리즘인 MQ(Multi-queue) 및 ARC 알고리즘 보다 우수한 성능을 보이며, X-RAY 등의 배타적 캐시 알고리즘 (Exclusive Caching) 수준의 성능을 보이고 있음을 입증한다.

CURRICULUM VITAE

EDUCATION

- Ph.D. in Computer Science and Engineering, Pohang University of Science and Technology (Mar. 2005 - Feb. 2011)
- M.S. in Computer Science and Engineering, Hanyang University (Mar. 2002 - Feb. 2004)
- B.S. in Chemical Engineering, Hanyang University (Mar. 1994 - Feb. 2002)

RESEARCH INTERESTS

- File and Storage Systems
- Real-time Operating Systems
- System Virtualization

Work Experience

- Researcher, POSTECH Information Research Laboratory (Mar. 2004 - Feb. 2005)

Teaching Experience

Lecturer

- IT System Administrator Program, Ansan Office of Education, Korea (Jul. 2003)
- Linux System Programming, Hanyang University, Korea (Jul. - Feb. 2002)
- Web Programming, Hanyang University, Korea (Oct. - Nov. 2001)
- Linux System Administrator Program, Hanyang University, Korea (Jul. 2001)
- IT System Administrator Program, Ansan Office of Education, Korea (Dec. 2000 - Jan. 2001)

Teaching Assistant

- Introduction to Programming (CSED 103), Division of Electrical and Computer Engineering, Pohang University of Science and Technology (Aug. - Dec. 2008)

Miscellaneous

- Compulsory Military Service, Sergeant, Republic of Korea Military Police (Dec. 1996 - Jan. 1999)

PUBLICATIONS

INTERNATIONAL JOURNAL PAPERS

1. **Woojoong Lee** and Chanik Park, "Reuse-distance Aware Block Management in Adaptive Replacement Cache", 2010 (In review)
2. **Woojoong Lee**, Shine Kim, and Chanik Park, "PosCFS+: A Self-managed File Service in Personal Area Network," ETRI Journal Vol 29. no 3, pp 281-291, 2007
3. Dongwook Kang, **Woojoong Lee**, and Chanik Park, "Kernel Thread Scheduling in Real-time Linux for Wearable Computers," ETRI Journal Vol. 29. no. 3, pp 270-280, 2007
4. **Woojoong Lee**, Shine Kim, Jonghwa Shin, and Chanik Park, "PosCFS: An Advanced File Management Technique for the Wearable Computing Environment," Lecture Notes in Computer Science 4096, pp 965-975, Proc. EUC'06, IFIP, 2006

INTERNATIONAL CONFERENCE PAPERS

5. **Woojoong Lee** and Chanik Park, "An Adave Chunking Method for Personal Data Backup and Sharing," 8th USENIX FAST'10 poster session, 2010
6. Yongseok Oh, **Woojoong Lee**, Wooram Park, Sejin Park, and Chanik Park, "PosFFS2: A New NAND Flash Memory File System Supporting Snapshot in Embedded Linux," 8th USENIX FAST'10 poster session, 2010
7. **Woojoong Lee**, Young-Ki Hong, and Chanik Park, "An Agent Framework for CE Devices to Support Storage Virtualization on Device Ensembles," 6th Annual IEEE Consumer Communications and Networking Conference (CCNC 09), 2009
8. Baekjae Sung, Sejin Park, **Woojoong Lee**, and Chanik Park, "Enhancing Robustness of an iSCSI-based File System in Wireless Networks," Int'l Symp. on Frontiers in Computer Architecture Design, 2008
9. **Woojoong Lee** and Chanik Park, "An Event Notification Framework for PosCFS+ Distributed File Service in Personal Area Network," International Conference On Next-Generation Computing, 2007

10. Injung Kim, Min Kyung Hwang, **Woojoong Lee**, and Chanik Park, “u-PC: Personal Workspace on a Portable Storage,” ACM Mobility Conference, 2007

11. Dongwook Kang, **Woojoong Lee**, and Chanik Park, “Dynamic Kernel Thread Scheduling for Real-Time Linux,” Real-time Linux Workshop, 2006

12. Seungjun Shim, **Woojoong Lee** and Chanik Park, “An Efficient Snapshot Technique for Ext3 File System in Linux 2.6,” Real-time Linux Workshop, 2005

13. Jonghwa Shin, **Woojoong Lee**, Shine Kim and Chanik Park, “PosCFS : A Context-aware File Service for The Wearable Computing Environment,” Proceeding of Next Generation Personal Computer (NGPC), 2005

DOMESTIC JOURNAL PAPERS

14. 박찬익, **이우중**, 성백재, “모바일 데스크탑 환경을 위한 운영체제 가상화 기술,” 정보과학회지 26권 106호, pp. 67 75, 2008

15. **이우중**, 홍영기, 박찬익, “PosCFS+: 모바일 컴퓨팅 환경에서의 디바이스 연계통합을 위한 분산 파일 서비스,” 한국 차세대 컴퓨팅학회 논문지 4권 3호, 2008

16. 성백재, 황민경, 김인정, **이우중**, 박찬익, “Wireless u-PC: 무선 네트워크 스토리지를 이용한 개인 컴퓨팅 환경의 이동성을 지원하는 서비스,” 정보과학회논문지: 컴퓨팅의 실제 및 레터 제14권 제9호, pp.916 920, 2008

17. **이우중**, 박찬익, “An Event Notification Framework for PosCFS+ Distributed File Service in Personal Area Network,” 한국차세대컴퓨팅학회 논문지 3권 4호, pp 22 31 (invited from NGPC 2007), 2007

18. **이우중**, 황우식, 김정선, “A Remote Control System using Bluetooth,” 전자공학회 논문지 41권 CI편 2호, pp 79 88, 2004

19. **이우중**, 김정선, “A Performance Monitoring System for Heterogeneous SOAP Nodes,” 정보과학회 논문지 CP12, pp 484 490, 2004

DOMESTIC CONFERENCE PAPERS

20. 홍영기, **이우중**, 박찬익, “SSD와 HDD에 대한 리눅스 I/O Subsystem의 성능 평가,” 35회 정보과학회 추계학술대회, pp 260 264, 2008

21. 성백재, 박세진, **이우중**, 박찬익, “무선네트워크 환경에서 iSCSI 기반 파일시스템의 안정성 향상기법,” 35회 정보과학회 추계학술대회, pp 352 357, 2008

22. 성백재, **이우중**, 박찬익, “WIRELESS U-PC: 무선 네트워크 스토리지를 이용한 개인 컴퓨팅 환경 이동성 지원 서비스,” 차세대컴퓨팅학회 추계학술대회, 2008

23. 홍영기, **이우중**, 박찬익, “휴대용 IT 기가 상 데이터 통합관리를 위한 PosCFS+ 파일시스템 에이전트,” 차세대컴퓨팅학회 추계학술대회, 2008

24. 신종화, **이우중**, 박찬익, 조일연, 한동원, “A Distributed File Service for Wearable Computing Environment,” Korea Computer Congress, 2005

25. **이우중**, 황우식, 김정선, “블루투스를 이용한 가전기기 원격제어 시스템,” 한국 정보과학회 추계학술대회, 2003

PATENTS

1. Chanik Park, **Woojoong Lee**, Shine Kim, “FILE SERVICE SYSTEM IN PERSONAL AREA NETWORK,” US, application number, 11/869223, Application date: Oct.10, 2006

2. Chanik Park, **Woojoong Lee**, Shine Kim, “유비쿼터스 환경에서 데이터 통합관리를 위한 분산 파일 서비스 방법 및 시스템,” Korea, Publication number: 10-0932642, Publication date: Dec. 10, 2009

3. Chanik Park, Seokgan Jung, Kyunghoon Lee, **Woojoong Lee**, Injung Kim, Min-kyung Hwang, “iSCSI와 UPnP 표준을 이용한 가상 응용 수행 지원장치 및 이 장치의 이용방법 ,” Korea, Publication number: 10-10-0869726, publication date: Nov. 14, 2008

4. Chanik Park, **Woojoong Lee**, Dongwook Kang, “커널 스레드에 의한 실시간 태스크의 응답시간 증가를 방지하는 동적 스케줄링 기법 ,” Korea, Application number: 10-2006-95304, Application date: Sept. 29, 2006

BOOK

1. Debian GNU/Linux (리눅스 서버 최강자 데비안 GNU/Linux), Hanbit media Inc., co-authored with Debian user group, Oct 28, 2001

RESEARCH PROJECTS

1. UFD 기반 소프트웨어 이동 기술을 위한 응용 수행 환경 가상화 지원 기술 개발 , 삼성전자, Sept. 2007.09 - Feb. 2008
2. 응용프로그램 이동성 지원을 위한 무선 Smart Application Device 개발, 삼성전자, Mar. 2007 - Feb. 2008
3. 핀홀 검출을 위한 임베디드 리눅스 기반 고정밀 영상처리 시스템 개발 , 산업자원부, Mar. 2005 - Feb. 2007
4. 웨어러블 퍼스널 스테이션 개발 , 정보통신부, Mar. 2004 - Feb. 2008
5. 블루투스 기반 가전기기 원격 제어 시스템에 관한 연구, 덕진전자, Mar. 2002 - Jun. 2003