



# Applied Computing Review

**Dec. 2015, Vol. 15, No. 4**

## Frontmatter

<b>Editors</b>		<b>3</b>
<b>SIGAPP FY'15 Quarterly Report</b>	J. Hong	<b>4</b>
<b>SAC 2016 Progress Highlights</b>	H. Haddad	<b>5</b>

## Selected Research Articles

<b>Multiple Protocol Transport Network Gateway for IoT Systems</b>	C. Shih and G. Wu	<b>7</b>
<b>Dynamic System Modeling of Evolutionary Algorithms</b>	G. Šourek and P. Pošík	<b>19</b>
<b>Improving Random Write Performance in Homogeneous and Heterogeneous Erasure-Coded Drive Arrays</b>	N. Jeremic, H. Parzyjegl, and G. Mühl	<b>31</b>
<b>Optimizing Swap Space for Improving Process Response after System Resume</b>	S. Lo, H. Lin, and Z. Chen	<b>54</b>

# Applied Computing Review

Editor in Chief

Sung Y. Shin

Associate Editors

Hisham Haddad  
Jiman Hong  
John Kim  
Tei-Wei Kuo  
Maria Lencastre

## Editorial Board Members

Rafael Accorsi  
Gail-Joon Ahn  
Rachid Anane  
Davide Ancona  
João Araújo  
Javier Bajo  
Giampaolo Bella  
Marcello Maria Bersani  
Albert Bifet  
Stefano Bistarelli  
Ig Ibert Bittencourt  
Emmanuel G. Blanchard  
Gloria Bordogna  
Jeremy S. Bradbury  
Barrett Bryant  
Antonio Bucchiarone  
Artur Caetano  
Alvin Chan  
Li-Pin Chang  
Seong-Je Cho  
Soon Ae Chun  
Juan Manuel Corchado  
Marília Curado  
Eitan Farchi  
Jose Luis Fernandez-Marquez  
Ulrich Frank  
Mário M. Freire  
João Gama  
Raúl Giráldez  
Karl M. Göschka  
Rudinei Goularte  
George Hamer

Hyoil Han  
Ramzi A. Haraty  
Jun Huang  
Yin-Fu Huang  
Angelo Di Iorio  
Seiji Isotani  
Takayuki Itoh  
Hasan Jamil  
Jinman Jung  
Soon Ki Jung  
Sungwon Kang  
Christopher D. Kiekintveld  
Bongjae Kim  
Dongkyun Kim  
Sang-Wook Kim  
Stefan Kramer  
Shonali Krishnaswamy  
Alberto Lluch Lafuente  
Paola Lecca  
Byungjeong Lee  
Maria Lencastre  
Hong Va Leong  
Peter Lewis  
João Lourenço  
Sergio Maffei  
Marjan Mernik  
Raffaella Mirandola  
Eric Monfroy  
Marco Montali  
Marco Di Natale  
Alberto Núñez  
Rui Oliveira

Barry O'Sullivan  
Ganesh Kumar P.  
Apostolos N. Papadopoulos  
Gabriella Pasi  
Anand Paul  
Manuela Pereira  
Ronald Petrlic  
Peter Pietzuch  
Maria da Graça Pimentel  
Beatriz Pontes  
Rui P. Rocha  
Pedro P. Rodrigues  
Juan Manuel Corchado Rodriguez  
Agostinho Rosa  
Davide Rossi  
Giovanni Russello  
Gwen Salaun  
Patrizia Scandurra  
Jean-Marc Seigneur  
Dongwan Shin  
Eunjee Song  
Marielle Stoelinga  
Junping Sun  
Francesco Tiezzi  
Dan Tulpan  
Julita Vassileva  
Teresa Vazão  
Mirko Viroli  
Wei Wang  
Denis F. Wolf  
Raymond Wong  
Stefano Zacchiroli

# SIGAPP FY'15 Quarterly Report

October 2015 – December 2015

Jiman Hong

## Mission

To further the interests of the computing professionals engaged in the development of new computing applications and to transfer the capabilities of computing technology to new problem domains.

## Officers

Chair	Jiman Hong Soongsil University, South Korea
Vice Chair	Tei-Wei Kuo National Taiwan University, Taiwan
Secretary	Maria Lencastre University of Pernambuco, Brazil
Treasurer	John Kim Utica College, USA
Webmaster	Hisham Haddad Kennesaw State University, USA
Program Coordinator	Irene Frawley ACM HQ, USA

## Notice to Contributing Authors

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library and in any Digital Library related services
- to allow users to make a personal copy of the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will refer requests for republication directly to you.

## Next Issue

The planned release for the next issue of ACR is March 2016.

## SAC 2016 Progress Highlights

The 31<sup>st</sup> Annual ACM Symposium on Applied Computing (SAC) will be held in Pisa, Italy, Monday April 4 to Friday April 8, 2016, in the Palazzo dei Congress Center in Pisa. The *Tutorials Program* is planned for Monday; the *Technical Program* for Tuesday through Friday; the *Student Research Competition (SRC) Program* for Tuesday (display session) and Wednesday (presentations session), respectively; and the *Posters Program* for Thursday.

SAC 2016 has received 1047 submissions, from 58 countries. The review process resulted in accepting 251 papers, leading to acceptance rate of 23.97% across all 37 tracks. In addition, approximately 90 posters will be invited for participation in the Posters Program. These are papers that have gone through the review process as papers. The SRC Program received 47 submissions. After the review process by the respected track committees, 22 SRC abstracts have been invited to compete during the SRC Program. The accepted abstracts will compete for three cash prizes (\$500, \$300, and \$200) and winners will be recognized during the banquet event on Thursday April 7, 2016. The first place winner can proceed to the National ACM SRC program. Furthermore, 12 tutorial proposals were received and reviewed by the organizing committee and 7 tutorials have been invited to participate in the Tutorials Programs. Details are posted on the conference website.

As planning is underway, information about hotels, transportation, excursions, and reservation forms are posted on the conference website (<http://www.acm.org/conferences/sac/sac2016/>). The local organizing committee recommends attendees to book their transportation and hotel rooms as early as possible. The registration system is now open. Included in the registration fee, SAC will provide daily lunches, coffee breaks, a reception on Tuesday in the cloister of Santa Maria del Carmine, and a banquet dinner on Thursday at the historic train station Stazione Leopolda. The Steering and Organizing committees are pleased to have SAC 2016 in the historic city of Pisa. We invite you to join us this April, meet other attendees, enjoy the conference programs, and have a pleasant stay in Pisa and Italy. We hope to see you there.

#	Track	Submissions	Accepted Papers
1	BIO	16	4
2	CC	53	13
3	CIVIA	24	6
4	DADS	20	5
5	DM	37	9
6	DS	16	4
7	DTTA	25	6
8	EADD	14	3
9	EC	11	3
10	EMBS	30	7
11	HCI	21	5
12	HEALTH	29	7
13	IAR	38	9
14	IILE	15	4
15	IRMAS	21	5
16	MCA	46	11
17	MUSEPAT	20	5
18	NC&DLCC	9	2
19	NET	19	4

#	Track	Submissions	Accepted Papers
20	OOPS	22	5
21	OS	40	10
22	PAPP	10	2
23	PL	38	9
24	RE	25	6
25	RST	10	2
26	SATTA	24	6
27	SE	102	24
28	SEC	58	14
29	SGST	13	3
30	SOAP	16	4
31	SONAMA	49	12
32	SP	24	6
33	SVT	56	13
34	SWA	29	7
35	TRECK	11	3
36	WCN	26	6
37	WT	30	7

#	Track	SRC Submissions
1	CC	1
2	DTTA	1
3	DS	1
4	EC	1
5	HCI	1
6	HEALTH	1
7	IAR	1
8	DM	1
9	MCA	3
10	NC	1
11	SE	4
12	SEC	1
13	SONAMA	1
14	SWA	1
15	SATTA	1
16	SVT	1
17	WCN	1

#	Country	SRC Participants
1	USA	3
2	Hong Kong	2
3	South Korea	5
4	Brazil	2
5	Greece	1
6	Tunisia	1
7	Italy	2
8	Ireland	1
9	India	1
10	Russia	1
11	Austria	1
12	Germany	1
13	UK	1

#	Country	Paper Submissions
1	Afghanistan	1
2	Algeria	2
3	Argentina	1
4	Australia	7
5	Austria	7
6	Barbados	2
7	Belgium	10
8	Brazil	249
9	Cameroon	1
10	Canada	29
11	Chile	2
12	China	32
13	Cuba	2
14	Czech Republic	12
15	Denmark	5
16	Egypt	1
17	Estonia	2
18	Finland	16
19	France	56
20	Germany	53
21	Greece	10
22	Hong Kong	3
23	India	30
24	Iran	9
25	Ireland	6
26	Israel	5
27	Italy	104
28	Japan	29
29	Jordan	1

#	Country	Paper Submissions
30	Kuwait	2
31	Latvia	2
32	Lebanon	7
33	Luxembourg	6
34	Macau	1
35	Malaysia	1
36	Morocco	4
37	Netherlands	19
38	New Zealand	1
39	Norway	11
40	Pakistan	2
41	Portugal	28
42	Republic of Korea	64
43	Réunion	1
44	Russian Federation	4
45	Saudi Arabia	2
46	Serbia	1
47	Singapore	5
48	Slovenia	1
49	Spain	20
50	Sweden	11
51	Switzerland	10
52	Taiwan	8
53	Thailand	1
54	Tunisia	37
55	Turkey	6
56	United Arab Emirates	8
57	United Kingdom	37
58	United States	60

On Behalf of SAC Steering Committee



Hisham Haddad

Member of SAC Organizing and Steering Committees

# Multiple Protocol Transport Network Gateway for IoT Systems

Chi-Sheng Shih

Graduate Institute of Networking and Multimedia  
Department of Computer Science and  
Information Engineering  
National Taiwan University, Taipei, Taiwan  
cshih@csie.ntu.edu.tw

Guan-Fan Wu

Graduate Institute of Networking and Multimedia  
Department of Computer Science and  
Information Engineering  
National Taiwan University, Taipei, Taiwan  
brotherwin2002@yahoo.com.tw

## ABSTRACT

Exchange data among devices in machine-to-machine (M2M) and Internet-of-Things (IoT) systems is an essential feature for these systems. For the sake of simplicity and reliability, many M2M and IoT solutions support homogeneous/single network deployment. However, when the complexity of the system and the number of device vendors continue to grow, it is challenging, if not impossible, to build M2M and IoT systems with devices using single communication protocol. The targeted problem of this paper is to enable the communication among devices in heterogeneous networks. Multiple Protocol Transport Network (MPTN) gateway is designed to be a distributed messaging gateway to enable messaging among multiple networks. To leverage the routing capability in existing network protocols, MPTN gateway converts an end-to-end message request to a multiple segment message based on network topology. The protocol schedules periodic routing table update to pro-actively keep the routing table up to dates with long time intervals, and allows on-demand route update to shorten the delay on topology and connectivity change. The performance evaluation results show that the protocol completes table update within tens of milliseconds when there are more than ten devices in a system.

## CCS Concepts

•Computer systems organization → Embedded systems; *Redundancy*; Robotics; •Networks → Network reliability;

## Keywords

ACM proceedings

## 1. INTRODUCTION

The development on embedded systems introduce highly automatic and/or intelligent computation services into practice. It allows autonomous computation systems to communicate with each other and to collaborately accomplish the works requiring large amount of information and intelligence. Many existing M2M systems are designed to work

in centralized manner and homogeneous networks. Smart Things<sup>1</sup> is one example. Smart Things hub serves the communication gateway of all Smart Things devices and all the sampled data are sent to Smart Things servers to process. Using a single communication protocol has its own merits. For instance, ZigBee can support low bit-rate communication with very low energy consumption; optical communication can support gigabit transmission within a short range. However, it is a challenge to federate the subsystems using different communication protocols so as to seamlessly collaborate.

When more and more sensing and control devices are designed to meet different requirements and support different use scenario, it is inevitable for the devices in M2M and IoT systems to communicate with other devices via heterogeneous communication systems. Although several M2M and IoT solutions support heterogeneous networks, static and centralized approach is used. Consequently, the system suffers from single point failure, flexibility, and scalability. This paper presents the design and implementation of a distributed and dynamic meta-routing mechanism on multiple network transport gateway. The proposed mechanism leverages the transport layer and beyond of existing network protocols, and compose the routes of heterogeneous networks protocols in distributed manner.

To support heterogeneous network communication, a communication gateway that deal with the message forwarding on top of transport layer of OSI model is designed and is called *Multiple Protocol Transport Network* gateway. The gateway receives the messages from applications, selects a network interface to send the message, and passes the message to transport layer. The meta-routing mechanism selects an proper network interface to next hop in heterogeneous networks and considers the selected next hop as destination. Device insertion, deletion, and failure handling mechanisms are all implemented in the gateway.

This paper is organized as follows. Section 2 presents the background of routing protocols in ad hoc networks and the related researches on multiple network protocols. Section 3 presents the architecture of the WuKong middle-ware and the problem of interest. Section 4 presents the design of multiple-protocol network gateway and distributed meta-routing mechanism. Section 5 presents the results of per-

Copyright is held by the authors. This work is based on an earlier work: RACS'15 Proceedings of the 2015 ACM Research in Adaptive and Convergent Systems, Copyright 2015 ACM 978-1-4503-3738-0. <http://dx.doi.org/10.1145/2811411.2811532>

<sup>1</sup><http://www.smartthings.com>

formance evaluation. Section 6 concludes the paper.

## 2. BACKGROUND AND RELATED WORKS

This section presents the works on routing protocols for ad hoc networks and multiple protocols supports. Routing protocols for ad hoc networks can be classified into two categories: *pro-active protocols* and *reactive protocols*. Pro-active protocols maintain complete routing table for all the nodes all the time. Hence, pro-active protocols repeatedly update the routing table. Destination-Sequenced Distance-Vector (DSDV) is one example of pro-active protocols. Reactive protocols update the routes when source node needs to send data, and each node keeps routing table for a predefined time interval. Ad hoc On-demand Distance Vector (AODV) is one example of reactive protocols. The following presents the DSDV and AODV protocols for ad hoc networks.

DSDV protocol was developed by C. Perkins and P. Bhagwat in 1994 [7]. It solved the routing loop problem in ad hoc network. A sequence number is generated by the source node when the network topology changes. The source node sends the update message with this sequence number. The sequence number is an even number when the link is present; otherwise the sequence number is an odd number when the link is absent. DSDV protocol keeps the routing information on every node. The nodes exchange and update their routing information periodically. This feature generates routing traffic even the node has no data to send, and may shorten the battery life of sensor nodes. Because DSDV maintains complete network routing information on all the nodes, it is suitable for a network which has small number of nodes.

Figure 1 shows an example ad hoc network for DSDV protocol. In DSDV protocol, every node stores its complete routing table. The routing tables for the node  $N_a$  and  $N_b$  in the example are shown in Table 1. Each entry in the table keeps the cost (metric) and next hop on the route to reach one destination. For example, if node  $N_a$  wants to send a message to node  $N_e$ , the packet should be routed to node  $N_c, N_d$ , and  $N_e$  in order. The message will take three hops to reach its destination node  $N_e$  and the next hop on the routing path is node  $N_c$ .

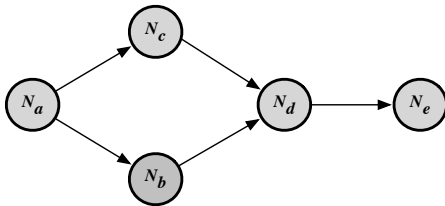


Figure 1: An example ad hoc network

AODV protocol was developed by C. Perkins and E. Belding-Royer in 1999 [6]. Similarly, AODV keeps track of the routing information to every destination in routing table, and sequence numbers to avoid routing loop and to verify if the information in routing table are up-to-date. If a route has not been used to transmit packets, the route will be only valid for a predefined time interval. When the interval ex-

Table 1: Routing Tables in DSDV protocol

Routing table on node $N_a$			Routing table on node $N_b$		
Dest.	cost	next	Dest.	cost	next
$N_a$	0	$N_a$	$N_a$	1	$N_a$
$N_b$	1	$N_b$	$N_b$	0	$N_b$
$N_c$	1	$N_c$	$N_c$	2	$N_a$
$N_d$	2	$N_c$	$N_d$	1	$N_d$
$N_e$	3	$N_c$	$N_e$	2	$N_d$

Routing table on node $N_c$		
Destination	cost	next
$N_a$	1	$N_a$
$N_b$	2	$N_a$
$N_c$	0	$N_c$
$N_d$	1	$N_d$
$N_e$	2	$N_d$

pires, an expiration event will be sent to its neighbors to invalidate the route entry. AODV creates a route before sending packet. AODV creates one route by query and reply message between source and destination nodes. All the intermediate nodes store the route information in their routing tables in corresponding entry. Hence, it has no communication overhead on maintaining routing table when one node has no data to send, and therefore it is suitable for highly dynamic networks.

There are three types of control messages to establish a route: Route Request Message (RREQ), Route Reply Message (RREP), and Route Error Message (RERR). RREQ is a route request packet which is broadcast to the network when a route is not available. RREP is a route reply packet which is sent by the nodes on selected path and to the source node. RERR is a route error packet which is broadcasted to neighborhood when any node detects a route is invalid.

Figure 2 demonstrates how to establish AODV routes. There are five nodes in the figure, including source node  $N_s$ , destination node  $N_d$ , three intermediate nodes  $N_a$ ,  $N_b$ , and  $N_c$ . Source node  $N_s$  requests to send a packet to destination node  $N_d$ . In Step ①:  $N_s$  broadcasts RREQ message to its neighbor  $N_a$  and  $N_b$ . In Step ②,  $N_a$  and  $N_b$  broadcast message RREQ again to their neighbor  $N_d$  and  $N_c$ . In Step ③,  $N_c$  broadcasts message RREQ again and destination node  $N_d$  replies RREP to  $N_c$ . In Step ④, node  $N_a$  replies RREP to  $N_s$ . Finally, node  $N_s$  establishes the route  $N_s \rightarrow N_a \rightarrow N_c \rightarrow N_d$ .

Yoon, Wonyong and Vaidya [12] proposed a routing mechanism in multiple interfaces. There are two types of interfaces: a primary 802.11a interface and a secondary 802.11b interface. 802.11a interface has a high throughput but a shorter transmission range. 802.11b has a lower throughput but a longer transmission range. Under normal conditions, a TCP flow uses a primary path over the 802.11a interface discovered by a reactive routing protocol. In presence of route breakage due to node mobility, it restores its backup path over the 802.11b interface which is already maintained by a pro-active routing protocol and is being used for delivery of control or management packets.

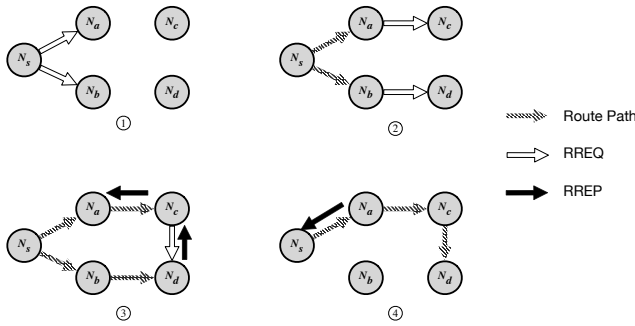


Figure 2: Route Establishment for AODV

### 3. SYSTEM ARCHITECTURE AND PROBLEM DEFINITION

The grand vision for WuKong middle-ware is that future IoT should have "zero-cost" deployment where users of an IoT application do not need to be concerned on how and where to deploy sensors. The built-in intelligence from the proposed IoT support can automatically perform the optimal sensor node configuration, bandwidth allocation, fault management, and re-configuration in response to new missions and new device deployment. Much like the past transition from low-level assembly codes to high-level programming using general purpose OS and compiler support, IoT programming may be as platform-independent as possible while keep only the most essential system primitives to allow automatic performance optimization.

WuKong middle-ware consists of two major components to fill the gap for developing and managing M2M and IoT applications: *intelligent middle-ware* and *flow-based developing framework*. WuKong middle-ware is regarded as a Virtual Middle-ware (VMW). The reasons for this are two-folds. First, as sensor networks become widely available, it is very likely that applications have to use sensors developed by different manufactures and communicating with different network protocols. Having a virtual sensor will allow applications to run on heterogeneous networks of sensors. Second, when the system decides to reconfigure the network, the process of reprogramming nodes will be less expensive by using a virtual machine design so that line of codes will be less since the virtual sensor can offer higher level primitives specific for IoT applications. On top of WuKong middle-ware, WuKong framework can postpone binding logical components with physical devices until an application is deployed, rather than when an application is developed. With the intelligent middle-ware and FBP, WuKong framework enables intelligent binding for IoT systems.

WuKong middle-ware is developed as open source project and is available for the public at <https://github.com/wukong-m2m/wukong-darjeeling>. At time of this writing, the ports for Intel Galileo, Intel Edison, and Arduino-compatible devices are available.

#### 3.1 System Architecture of WuKong Middle-ware

WuKong system is a distributed computing runtime to ac-

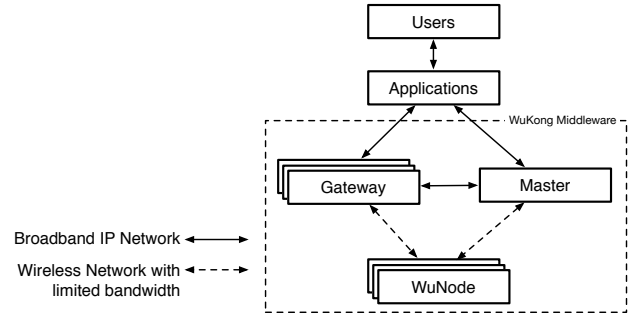


Figure 3: System Architecture of WuKong

complish the requests from users and applications. Figure 4

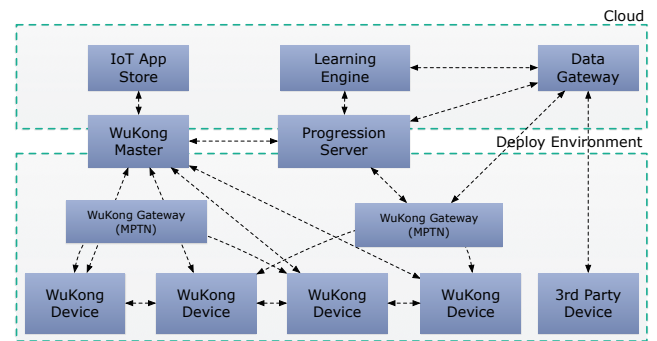


Figure 4: System Architecture of WuKong

shows the system architecture of WuKong middle-ware. WuKong systems consist of the components on the cloud and that in deployment environment. On the cloud server, WuKong deploys IoT application store, learning engines to learn users' behaviors/contexts, and data gateway to archive historical data. In deployment environment, WuKong deploys one WuKong master (called WuMaster for short), several WuKong gateways, and number of WuKong devices. The devices in deployment environment are described below.

- **WuMaster:** The most important task of WuMaster is to configure, optimize, and reconfigure sensors. To do this, it communicates with sensors through a layer of abstraction, hiding hardware and network details of the underlying sensor platform. During the discovery and identification phase, WuMaster uses the profile framework to discover the capabilities of connected sensors, and configure sensors' parameters. WuMaster is also responsible for managing the user-defined services in the system including deploying the service to the devices, making in-situ decision for software upgrade and service remapping. In practice, WuMaster will be deployed on a computational power and robust device which is capable of reliably receiving user requests and managing the services.
- **WuGateway:** WuGateway has two major responsibilities: *communication gateway* and *backup master*.

As a communication gateway, it has the capability of discovering devices, forwarding messages, and dispatching messages in heterogeneous networks. Communication gateway is named Multiple Protocol Transport Network (MPTN) gateway and will be discussed in this paper. In many deployment scenario, there can be several MPTN gateways, each of which uses different communication protocols to connect to the devices. As a backup master, it has the capability of replicating the state information and services on WuKong Master. When the WuKong Master is down or disconnected, the gateway can replace WuKong Master to manage the devices and services in the system.

- **WuDevices:** WuKong devices, shorted as WuDevices, represent the networked physical devices in the system. One WuDevice can be a combination of sensors, actuators, and computation services. To be part of the WuKong systems, a WuDevice should register itself to WuKong master directly or via WuGateway, identify its own capability via its profiles, and join the system. The services including sensing, control, and computation carried out on WuDevices are deployed by WuKong master. WuKong master deploys the service in the form of interpretive commands, native code or Java byte code, depending on the capability of the WuDevices. Figure 5 shows the software stack on WuDevices. NanoKang runtime is the virtual environment to carry application workload. It consists of boot-loader, communication subsystems, Java virtual machine and native profiles. Among these components, native profiles keep the capabilities of hardware devices. Native profiles and virtual profiles are parts of WuKong profile framework (WKPF), which are will presented later. WuClass represents pre-defined computation service in WuKong middle-ware. User-defined computation services are define by WuObject applications on top of WKPF.

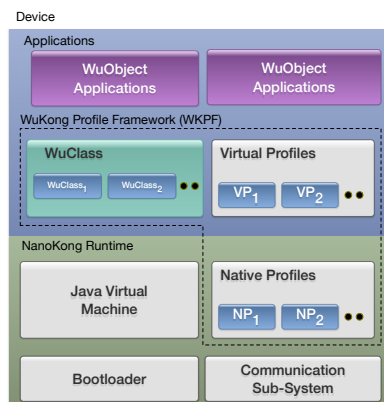


Figure 5: Software Stack on WuKong Devices

### 3.2 Tri-Framework Architecture

The life-cycle of IoT and M2M system is a loop and has no ends. It starts with the process of device discovery, continues with device identification, user need definition, service

composition, service deployment, and service/device update. After a service/device is updated, the process of device discovery repeats. The middle-ware for IOT and M2M take into account and support the life-cycle of such application scenario. In WuKong middle-ware, we develop a three orthogonal middle-ware frameworks to support this life-cycle:

1. **Profile framework:** to enable the handling of heterogeneous or new sensor nodes, and for high-level, logical abstraction of low-level, native sensor operations. WuKong profile framework (WKPF) is the core framework in WuKong to keep track of the properties of the devices, the value of properties, and links among computation components. Hence, on every device in WuKong systems, one WKPF must be deployed.
2. **Policy framework:** to allow user-friendly specification of application execution objective, and context-dependent optimization of IoT/M2M performance. Policies are applied when mapping logical components to physical devices.
3. **Progression framework:** to facilitate in-situ software upgrade for dynamically, progressive reconfiguration. WuKong progression framework is the core component for fault tolerance and program by examples to adapt to the change of the environment, device failure, and users' needs.

Our earlier works [9, 5, 10, 8, 11] show that WuKong middle-ware effectively enables run-time and remote reconfiguration, run-time service binding, dynamically service mapping, autonomous fault tolerance, and autonomous device management.

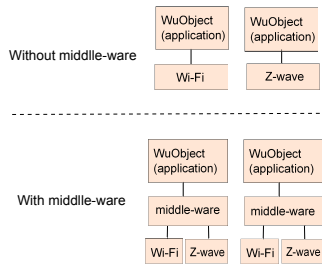
### 3.3 Targeted Problem: Heterogeneous Communication Gateway

In last few years, many IoT and M2M devices are equipped with different network interfaces, including Wi-Fi [2], Zig-Bee [4], BLE, Z-wave [3], Power-line Communication, and LoRa [1], for specific application domain and deployment environment. In addition, there are many attempts unify network interface for M2M and IoT systems. WuKong does not attempt to unify the network interface and recognize that the network interfaces in M2M and IoT will be heterogeneous to meet the communication requirement of various applications. Hence, the targeted problem of this work is to enable message transmission between devices using different network protocols without a centralized communication hub.

Many IoT/M2M systems use a centralized communication hub to resolve the targeted problem. One example is Smart Things hub, which support ZigBee, Z-Wave, and IP protocols<sup>2</sup>. Centralized communication hub approach has its own merits: *less expensive to deploy services* and *less expensive to maintain services*. On the other hand, the communication hub may become the performance and communication bottleneck when the number of devices in a system increases. Consequently, its scalability is limited.

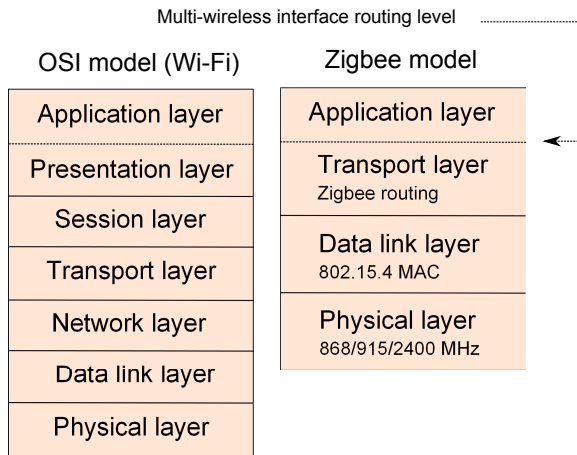
<sup>2</sup><https://shop.smarthings.com/#!/products/smarthings-hub>

Figure 6 presents the software architecture of targeted problem.



**Figure 6: Middle-ware between physical devices and application**

WuKong VM will be deployed to every WuDevice in the system. The middle-ware is located between application and physical devices. Multi-wireless interface routing system will select a communication interface to next hop, then the system takes the selected next hop as destination and send to network layer of selected communication interface. The physical device knows the packet destination but doesn't know the route destination. Transport layer routing is not conducted in the middle-ware and is left to individual communication interface.



**Figure 7: Multi-wireless interface routing level**

In M2M network, nodes may have one or more wireless communication interfaces (Wi-Fi, Zigbee, Z-wave) at the same time. One node can send data to each other indirectly, and never reject the forwarding request from other node. Node forwards the packet to next hop according to the routing path. The policy is specified by users or applications according users' need or context at run-time. It can be a throughput oriented policy to better utilize the throughput in the network, or an energy saving policy to reduce energy consumption in the system. Quality of Service (QoS) constraint can be maximal energy consumption or minimal transmission delay time during transmission. The network topology may change any time, for example: new node may come; old node may fail or change location.

According to the user policy, and network topology. The targeted problem is to design and implement a communication component to select proper communication interface to meet the performance requirement defined above. Furthermore, the interface selection has to be conducted dynamically according to run-time network topology.

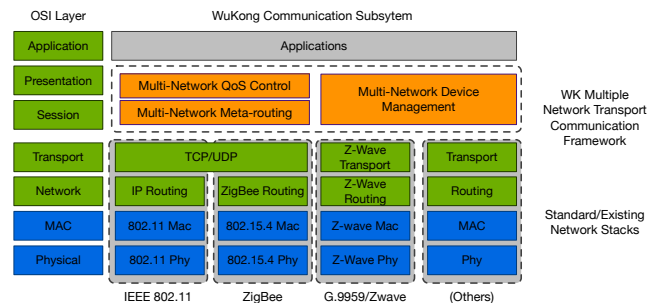
At time of writing, most of network routing protocols assume network interfaces are homogeneous. We design network routing mechanism across different network interfaces. The traditional network routing is designed for static environment, which is applicable for IoT. The new node having different network interface may join network at any time and any existing node may fail or change its location. Consequently, the network topology changes all the time, how to calculate a routing path and maintain routing path under dynamic network environment is the challenges of this work.

## 4. DESIGN AND IMPLEMENTATION OF DISTRIBUTED META-ROUTING MECHANISM

### 4.1 MPTN Gateway

MPTN gateway provides a message forwarding service for heterogeneous network messages in WuKong systems. A heterogeneous network message is a message whose source and destination network interface do not connect to same computer network. To broaden the applicability, MPTN gateway is designed as a software component to be deployed on WuKong devices which have multiple network interfaces, either homogeneous or heterogeneous. Further more, to take advantage of existing network stacks, MPTN gateway conducts meta-routing on top of transport layer, which selects an outgoing network interface for messages received from a different network interface and forwards the messages, for heterogeneous network messages.

Figure 8 presents the software architecture for MPTN gateway. MPTN gateway leverage transport, network, MAC,



**Figure 8: Software Architecture of MPTN Gateway**

and physical layer of existing network protocols. On top of transport layer, MPTN gateway implements three major components: *Device management*, *Meta-Routing* and *QoS Control*. Device management component manages the identification of WuDevices and network interfaces. Each WuDevice is assigned a unique ID among the devices managed by

one WuMaster. Meta-Routing component searches for the appropriate network interface to forward incoming messages. Last, QoS control component defines, stores, and applies the QoS policy to meta-routing. This paper aims on the design and implementation device management and meta-routing components.

In WuKong system, there are two types of messages. One is the *application message* and the other is the *routing message*. Application message is issued by applications to transmit data. It can be sampled data from the sensor, the commands to the actuator, or the reconfiguration message to the WuDevice. The routing message is issued by meta-routing component. Routing messages are the ones to update and maintain the routing table in the system. Figure 9 shows the flow of handling for IP messages in WuKong system. *WK-comm polling* is a component called by NanoKong, WuKong runtime, periodically to check if there is any message on network interface to be forwarded. If there is none, it returns to VM. If there is a routing message, the message will be forwarded to device handler to follow up. If there is an application message, the message will be forwarded to device handler to retrieve message from device buffer. Then, WuKong communication handler forwards the message to WuKong applications.

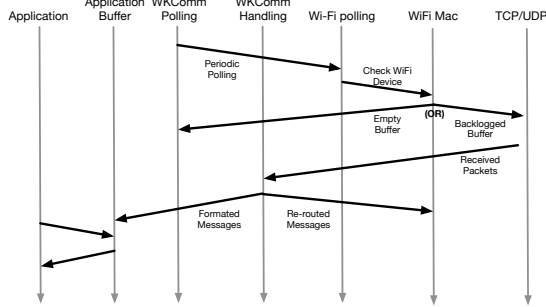


Figure 9: Flow of receiving message in WuKong

To manage the devices administrated by a WuMaster, each WuDevice will be assigned an identification unique among all WuDevices and each network interface will be assigned an identification unique among the network interface in same work. The unique device identification is called *global ID*, GID for short, and the unique network interface identification is called *local ID*, LID for short. A four bytes variable is allocated for each GID and a two bytes variable is allocated to each LID. Both GID and LID are assigned by WuMaster during device discovery.

Communication subsystem in both WuDevice and MPTN gateway route messages to its destination according to the routing table on the device. Every WuDevice in network maintains its own routing table, which keeps track of the routes to various destination devices. Each entry in routing table contains *Destination GID*, *Next hop GID*, *next hop LID*, *next hop interface*, *session ID*, and *route score*. Destination GID represents the GID of destination WuDevice. Next hop GID represents the GID of the next hop to the destination node. Next hop LID is the identification of network interface used to send message to next hop. Session

ID is used to identify the session of this row. Route score represents the score of one route. The score can be the energy consumed to transmit one byte of data or transmission efficiency, depending on scoring policy. For instance, energy saving policy uses energy utilization to evaluate a route and represents indicates the amount of consumed energy to send one byte of data. Hence, the smaller energy utility the better. On the other hand, throughput policy may use transmission efficiency to evaluate a route. The unit of transmission efficiency can be amount of time for transmitting one byte of data.

Figure 10 shows an example of WuKong network. There are four WuDevices in the network. The orange block inside WuDevice stands for Wi-Fi interface, and yellow block stands for Z-wave interface. The number labeled on each box represents LID of the interface. For example, WuDevice  $W_2$  has one Z-wave interface whose LID is 12, and no Wi-Fi interface. Without losing generality, the GID of WuDevice  $W_1, W_2', \dots$  are assigned 1, 2, ..., etc.

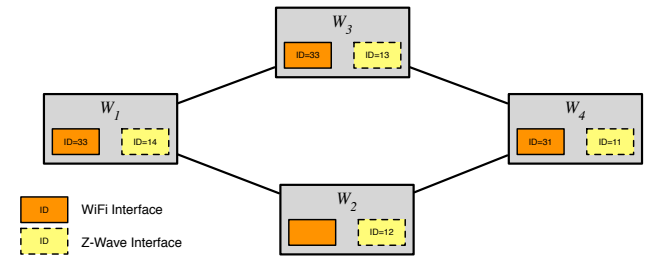


Figure 10: An example network of WuDevices

Table 2 shows the routing table on WuDevice  $W_1$ . In this example, the routing to WuDevice  $W_2$  selects Z-wave interface on WuDevice  $W_1$ ; the route to WuDevice  $W_3$  selects the Wi-Fi interface on WuDevice  $W_1$ . There is no direct link between WuDevice  $W_1$  and WuDevice  $W_4$ . Hence, the route to WuDevice  $W_4$  selects WuDevice  $W_2$  as the next hop and its interface is Z-wave on WuDevice  $W_1$ .

Table 2: Routing Table on WuDevice

Destination GID	Next-hop GID	Next hop LID	Next hop interface	Session ID	route score
1	-	-	-	-	-
2	2	12	Z-wave	1	0.2
3	3	33	Wi-Fi	1	0.25
4	2	12	Z-wave	2	0.45

## 4.2 Meta-Routing Table Update

### 4.2.1 Node Insertion and Deletion.

To insert new nodes to the network, node  $N_x$  will broadcast two types of routing messages: one is *route scoring message* and the other one is *table update message*. When node  $N_x$  joins the network, it broadcasts a route scoring message (RSM) to construct its routing table. The message consists of current route scores and previous hop of all the routes in

the table, which can be empty when joining the network for the first time. When the neighbors receive routing table of node  $N_x$  in RSM, the node calculates a new route score by adding/subtracting one hop score to/from the current score. The route score will be updated only if the new route score outperforms the current route score. Otherwise, it will keep the previous route score.

In the second step, node  $N_x$  broadcasts a table update message (TUM) to all its neighbors to collect latest routing table. While receiving TUM, the node replies its routing table to node  $N_x$ . To avoid duplicated table update, TUM will be transmitted for one hop only. Hence, the communication sub-system, including that on MPTN gateway and WuDevice, does not forward RSM. Node  $N_x$  waits for a time-out interval to collect the replies from its neighbors. After the timeout period, node  $N_x$  constructs its own routing table according to neighbors' replies. For each entry having same destination GID, node  $N_x$  keeps the entry whose the route score is the minimal/maximal, and discards the other replies. After constructing the routing table, node  $N_x$  periodically checks the routing table. If the routing table changes, node  $N_x$  broadcasts the changed entries in routing table to the network.

Every other reachable node updates the routes to node  $N_x$  on its routing table. Due to multi-path broadcast, the node may receive more than one route scoring message (RSM) request from node  $N_x$ . Hence, the node selects the best, either minimal or maximal, route score to update its local routing table. While updating the table, it collects destination GID, session ID, and route score from RSM request. In particular, previous hop GID information in RSM request will be inserted to local routing table as next hop GID. When a node receives TUM request from node  $N_x$ , it replies its full routing table.

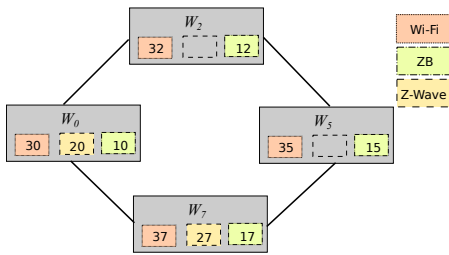


Figure 11: The example of network topology

Figure 11 shows the network topology for following examples. In this topology, WuDevice  $W_2$ ,  $W_5$  and  $W_0$  are already registered to WuMaster. WuDevice  $W_7$  is the new node. Table 3 shows the routing table on WuDevice  $W_0$  before WuDevice  $W_7$  joins the network. Figure 12 shows the process of registering a new node to WuMaster. In the first step, WuDevice  $W_7$  broadcasts RSM request to its neighbors (WuDevice  $W_0$  and  $W_5$ ), shown as Step 1.1 in Figure 12. After receiving RSM request, WuDevice  $W_0$  and  $W_5$  broadcast again the request to their neighbors (WuDevice  $W_2$ ), shown as Step 1.2 in Figure 12. The broadcast will stop when the node receives the same request. Finally, WuDevice  $W_2$  replies the message to its neighbor (WuDevice  $W_0$  and WuDevice  $W_5$ ), but WuDevice  $W_0$  and  $W_5$  will ignore

Table 3: Routing Table on WuDevice  $W_0$  before WuDevice  $W_7$  join network

Destination GID	Next-hop GID	Next hop LID	Next hop interface	Session ID	Route score
0	-	-	-	-	-
2	2	32	Wi-Fi	13	15
5	2	32	Wi-Fi	16	25

the request, shown as Step 1.3 in Figure 12. In second step, WuDevice  $W_0$ ,  $W_2$  and  $W_5$  update their routing tables according to the routing table in RSM request in first step. Table 4 shows the routing table on WuDevice  $W_0$  after second step. In third step, WuDevice  $W_7$  broadcasts TUM request

Table 4: Routing Table on  $W_0$  after  $W_7$  joins network

Destination GID	Next-hop GID	Next hop LID	Next hop interface	Session ID	Route score
0	-	-	-	-	-
2	2	32	Wi-Fi	13	15
5	2	32	Wi-Fi	16	25
7	7	37	Wi-Fi	22	10

to its neighbors (WuDevice  $W_0$  and  $W_5$ ) for their routing table. In forth step, WuDevice  $W_0$  and  $W_5$  reply their routing tables to WuDevice  $W_7$ . In fifth step, WuDevice  $W_7$  selects the route with minimal route score from replies, and then updates its own routing table.

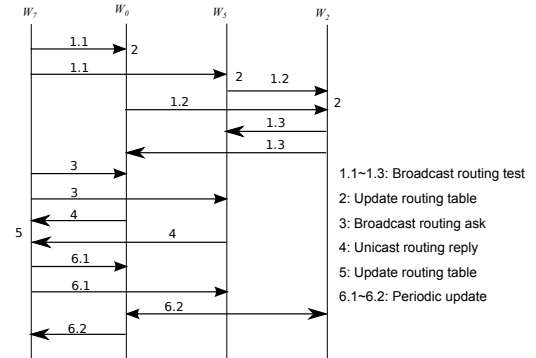
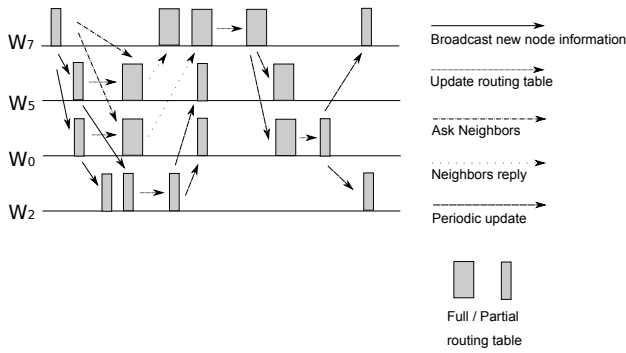


Figure 12: The procedure for node insertion

In sixth step, WuDevice  $W_7$  learns its routing table has been changed and, hence, broadcasts the changed routing table to its neighbors (i.e., WuDevice  $W_0$  and  $W_5$ ), shown as Step 6.1 in Figure 12. According to the updates, WuDevice  $W_0$  learns the route including WuDevice  $W_7$  and  $W_5$  is better than the route including WuDevice  $W_2$  and  $W_5$ . Finally, WuDevice  $W_0$  learns that its routing table has been changed and, hence, broadcasts the updated routing table to its neighbors, i.e., WuDevice  $W_2$  and  $W_7$ , shown at Step 6.2 in Figure 12.



**Figure 13: The time sequence of adding new device  $W_7$**

Figure 13 shows the time sequence of adding WuDevice  $W_7$ . In step 1: WuDevice  $W_7$  sends new node information (partial routing table) to WuDevice  $W_5$  and WuDevice  $W_0$ , and then WuDevice  $W_5$  and  $W_0$  reply the message to WuDevice  $W_2$ . In step 2: WuDevice  $W_2$ ,  $W_5$  and  $W_0$  update their routing table from new node information. In step3 and 4: WuDevice  $W_5$  and  $W_0$  reply their complete routing table to WuDevice  $W_7$ . In step 5 and 6: WuDevice  $W_2$  updates its routing table, and sends periodic update message to WuDevice  $W_5$  and  $W_0$ . Because the routing information from WuDevice  $W_2$  already exist in WuDevice  $W_5$  and  $W_0$ , the update is ignored. WuDevice  $W_7$  creates its routing table from replied messages, and then sends full routing table while doing periodic update. WuDevice  $W_0$  finds a shorter route from the periodic message, so WuDevice  $W_0$  updates its routing table and then sends the shorter route information (partial routing table) in next periodic update. Finally, WuDevice  $W_0$ 's periodic update information doesn't affect any other node, the network finish adding a new node.

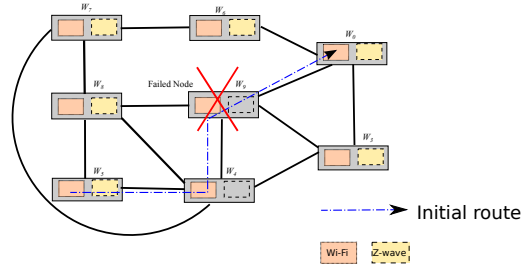
After step 6, WuDevice  $W_0$  knows that the shortest route is  $(W_0, W_7, W_5)_p$ . WuDevice  $W_0$  updates its routing table, and broadcasts the information to neighbor. Assume  $N_y$  is a node in network. If WuDevice  $W_0$  updates its routing table from WuDevice  $W_7$ , the route must be  $(W_0, W_7, N_y)_p$ . WuDevice  $W_0$  broadcasts the information to WuDevice  $W_7$ , the route  $(W_0, W_7, N_y)_p > (W_7, N_y)_p$ , WuDevice  $W_7$  will not update the routing table, therefore there is no updating loop.

#### 4.2.2 Routing Table Update for Node Failure.

The following presents are procedures to handle device failure in the network. Again, for the sake of presentation, destination node is represented by  $N_d$ , source node is represented by  $N_s$ , failed device is represented by  $N_f$ , the hop on the route before node failure is represented by  $N_e$ , and the failed interface on node  $N_f$  is represented by  $I_f$ .

When WuDevice  $N_e$  learns that it cannot forward messages to WuDevice  $N_f$ , WuDevice  $N_e$  records the GID of next hop

$N_f$  and its interface  $I_f$ . Then, WuDevice  $N_e$  broadcasts to ask its neighbor for how to reach  $N_d$ . WuDevice  $N_e$ 's neighbors look up their routing tables to find the route to node  $N_d$ , and then reply the next hop GID, next hop interface and route score on the route to WuDevice  $N_e$ . Last, WuDevice  $N_e$  checks neighbors' replies if there is any route whose next hop is different from WuDevice  $N_f$  or whose next hop interface is different from interface  $I_f$ . If exists, WuDevice  $N_e$  forwards the message to the neighbor which has the best route scores and meet these requirements. If there is none, WuDevice  $N_e$  will notify WuDevice  $N_s$  that the message cannot be sent and WuDevice  $N_d$  is not reachable.



**Figure 14: An example of routing table update for failure handling**

Figure 14 shows an example of routing table update for failure handling in WuKong gateway. In this example,  $W_5$  is the source node and  $W_0$  is the destination node.  $(W_5, W_4, W_9, W_0)_p$  is the route from  $W_5$  to  $W_0$ . However,  $W_9$  fails and is not reachable any more.

At first,  $W_5$  sends data to the next hop, i.e.,  $W_4$ .  $W_4$  search its routing table for the route to  $W_0$ , and learns that next hop is  $W_9$ .  $W_4$  tries to forward message to  $W_9$ , but  $W_9$  doesn't have any response. When  $W_4$  learns that  $W_9$  is unreachable.  $W_4$  records the GID of  $W_9$  and Wi-Fi interface.  $W_4$  broadcasts to its neighbors ( $W_5, W_8, W_3, W_7$ ) for how to reach  $W_0$ . After receiving the request from  $W_4$ , the neighboring WuDevices checks their routing tables, and reply the GID of next hop, next hop interface and route score for the route to  $W_0$ . The next hops replied by  $W_5, W_3, W_7$  and  $W_8$  are  $W_4, W_0, W_6$  and  $W_9$ , respectively. After collecting all these replies,  $W_4$  discards  $W_4$ (current node) and  $W_9$ 's Wi-Fi interface. The replies from both  $W_3$  and  $W_7$  can reach  $W_0$ . Since  $W_3$ 's score is better than  $W_7$ 's score,  $W_4$  selects  $W_3$  as next hop and recover the route to  $W_0$ . Next,  $W_4$  sends the table update message to its neighbors  $W_5, W_8, W_3, W_7$  in the following periodic update.  $W_3$  and  $W_7$  ignore the updating message because the route is not better than current route.  $W_5$  updates its routing table because the next hop to destination is  $W_4$ .  $W_8$  updates its routing table because the next hop to destination is a failed node.

Figure 15 presents the time sequence of handling WuDevice  $W_9$ 's failure.

#### 4.2.3 Periodic Routing Table Update.

To avoid the overhead of on-demand route update, every WuDevice will update its routing table periodically. If the table changes between the periods, it will broadcast the dif-

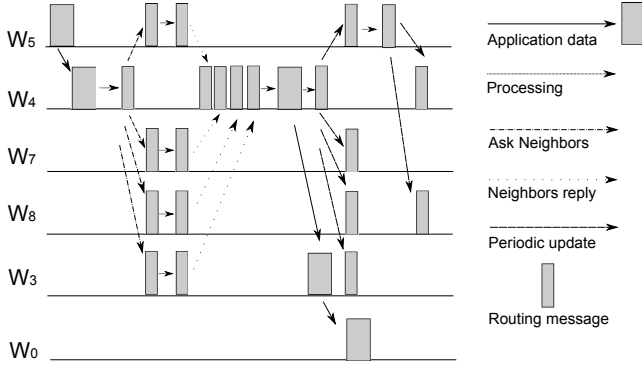


Figure 15: The time sequence of node fail

ference to its neighbors. The receiving WuDevice checks its routing table if any update is required. If any route in the update message is new to the receiving WuDevice, a new entry will be inserted to the routing table. Otherwise, it compares route score and next hop with its routing table. It updates the routing table in two cases. The first case is that the route score in update message is better than that in its routing table. The second case is that the next hop in routing table is the source node of update message. This means the next hop may discover a failed node and repair its routing table in this case. WuDevice should update its routing table because the update message comes from the next hop.

## 5. PERFORMANCE EVALUATION

### 5.1 Workload Parameters

To evaluate the performance of meta-routing mechanism, the experiments are set to evaluate two metrics. The first evaluation is the time overhead for fault recovery and routing table propagation delay. To evaluate the effectiveness of fault handling mechanism in MPTN gateway, three performance metrics are measured. *End-to-End message transmission time* refers to the time interval taken to send at source node and to receive a message at destination node when there is no fault. *Fault recovery delay* refers to the time interval from the time instance at which the fault is detected to that at which the message is forwarded to a recover route if exists. *Routing table propagation time* refers to time interval from the time instance at which the fault is detected to that at which the routing table being recovered on source node. The second evaluates the correlation between number of connected neighbors and fault recovery period.

The experiments were conducted on ten WuDevice Gen. 2 boards, which are Arduino-compatible devices. The devices have Z-Wave network interface on board and can install Arduino compatible Wi-Fi or ZigBee shields to support Wi-Fi or ZigBee network. When more than one network interface are installed, the device can serve as an MPTN gateway to forward messages between different network protocols. The period for updating routing table is set to 3000 ms and fault detection timeout is set to 3000ms.

The network topology of the experiment is shown in Figure 16. There are ten WuDevices in the network. WuDevice  $W_1$  is only equipped with Wi-Fi interface. WuDevice  $W_5$ ,  $W_7$  and  $W_8$  are only equipped with Z-wave interface. The other nodes are equipped with both Wi-Fi and Z-wave interface. Figure 16 also shows an example route before any WuDevice fails. In particular,  $(Z\text{-wave on } W_5, \text{ Wi-Fi on } W_4, \text{ Wi-Fi on } W_3)_p$  is the route to send messages from WuDevice  $W_5$  to Master.

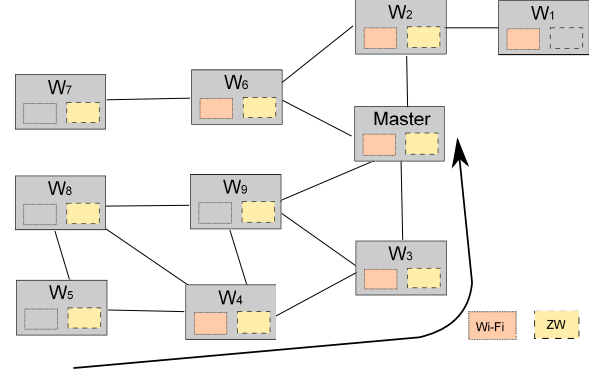


Figure 16: The network topology for experiment setting

The workload is a smart home application shown in Figure 17 in flow-based programming model. The application is designed to control the air conditioner in the room based on the temperature reading and occupancy of the room. If any user is presented in the room and the temperature is greater than threshold, the application will turn on air-conditioner. WuKong maps the applications to use motion sensors and infra-red sensors to detect whether the user is presented in the room or not. There are three rooms in the smart home. The sensors are connected to WuDevice  $W_7$ ,  $W_3$  and  $W_5$  respectively in three rooms. The data destinations are WuDevice  $W_1$ , WuMaster and  $W_9$  respectively in three rooms. Figure 16 also shows the route for the data flows:  $(W_7, W_6, W_2, W_1)_p$  is the message route for room  $R_1$ ,  $(W_5, W_4, W_9)_p$  is the message route for room  $R_2$ , and  $(W_3, \text{Master})_p$  is the message route for room  $R_3$ .

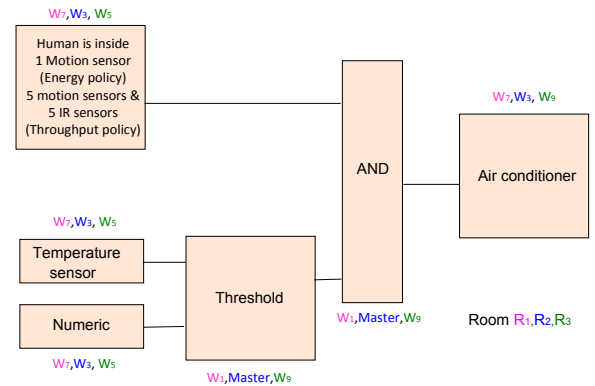


Figure 17: Example smart home application in flow base programming

## 5.2 Evaluation Results

To evaluate the effectiveness of fault handling mechanism in MPTN gateway, three performance metrics are measured. *End-to-End message transmission time* refers to the time interval taken to send at source node and to receive a message at destination node when there is no fault. *Fault recovery delay* refers to the time interval from the time instance at which the fault is detected to that at which the message is forwarded to a recover route if exists. *Routing table propagation time* refers to time interval from the time instance at which the fault is detected to that at which the routing table being recovered on source node.

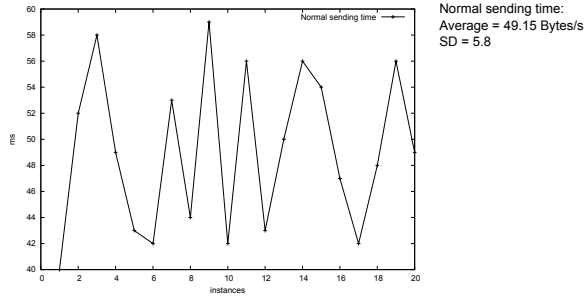


Figure 18: End-to-End Message Transmission Time

In this experiment, the average end-to-end delay for messaging is 49.15ms and its standard deviation is 5.8, which is 10% of the average value. The fault handling overhead for three different types of faults are evaluated in the experiments. In the first experiment, Wi-Fi interface on WuDevice  $W_3$  fails and, hence, the route from WuDevice  $W_4$ 's Wi-Fi interface to  $W_3$  is not available any more. WuDevice  $W_4$  broadcasts to ask neighbors through Z-wave. Four neighbors, i.e.,  $W_5$ ,  $W_8$ ,  $W_9$  and  $W_3$ , reply the routes to the master on their routing tables to WuDevice  $W_4$ . Figure 19 shows the results for handling the failure. The average fault recovery time is 3053.5 ms and standard deviation is 36.63. Note that among the fault recovery time, the gateway waits for the timeout period to learn that a device fails. Hence, it takes less than 100ms in average to recover the failure. The average routing table propagation delay in this experiments is 1552.55 ms and standard deviation is 790.43. While checking the log of the experiments, we find that the variation on propagation delay on table update was caused by the MAC control in Z-wave protocol.

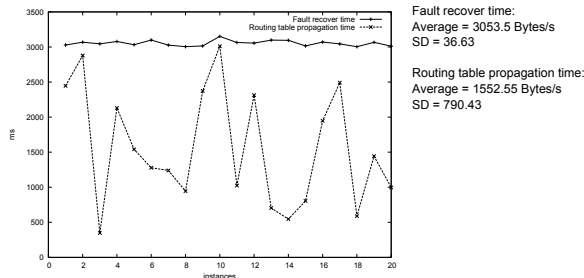


Figure 19: Fault recovery time and routing table propagation time when WuDevice  $W_3$ 's Wi-Fi interface fails

In second experiment, WuDevice  $W_3$  completely fails and cannot communicate with other nodes via both Wi-Fi and Z-Wave interface. Hence, the route from WuDevice  $W_4$  to  $W_3$  is not valid any more. After learning the fault, WuDevice  $W_4$  broadcasts to ask its neighbors for the route to WuMaster. Three neighbors, i.e.,  $W_5$ ,  $W_8$ , and  $W_9$ , reply their routes to WuMaster to  $W_4$ . Figure 20 shows the results in this case. The average fault recovery time is 3057.4 ms and standard deviation is 36.98. The average routing table propagation time is 1535.05 ms and standard deviation is 821.79.

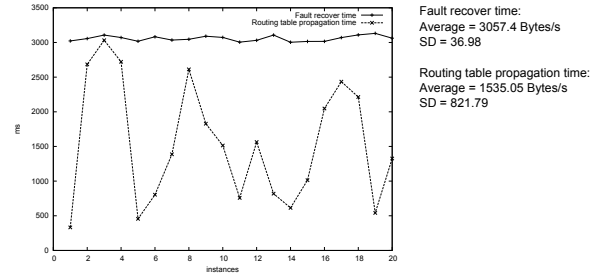


Figure 20: Fault recovery time and routing table propagation time when WuDevice  $W_3$  fails

In third experiment, WuDevice  $W_4$  fails. The route from  $W_5$  to  $W_4$  is not valid any more. Hence, WuDevice  $W_5$  broadcasts to ask its neighbor to find a new route. Figure 21 shows the results in this experiment. The average fault recovery time is 3055.4 ms and standard deviation is 38.4. The average routing table propagation time is 52.3 ms and standard deviation is 0.48. This case has the smallest routing table propagation time because the source node detects the fault and updates its routing table. These experiments show that all routing table propagation time is less than  $(N + 1) * (\text{update period})$ , where  $N$  is the number of hops to source node.

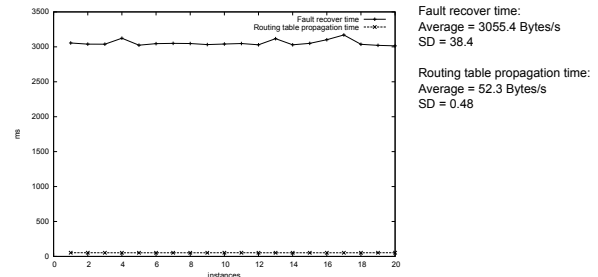


Figure 21: Fault recovery time and routing table propagation time when WuDevice  $W_4$  is failed

The second part of the experiments evaluate the correlation between the number of neighbors and the fault recovery overhead. A node uses Z-wave interface to broadcasts the asking packet to its neighbors and the neighbors reply to the node. We measure the time interval from the time the node send request to its neighbors until the time it receives the corresponding reply. The number of neighboring nodes is 5, 10, and 15. Figure 22 shows the results. In the first case, five neighbors reply messages to the node. The average time

is 290.45 ms and standard deviation is 15.52. In the second case, ten neighbors reply the message to the node. The average time is 562.5 ms and standard deviation is 21.62. In the third case, fifteen neighbors reply the message to the node. The average time is 852.4 ms and standard deviation is 22.74. According to the results, we observe that fault recovery period is closely related to the number of neighbors.

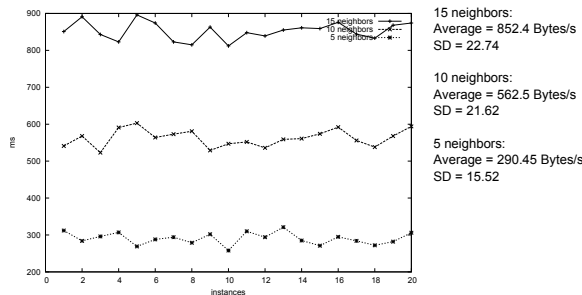


Figure 22: Fault Tolerance Overhead

## 6. CONCLUSION

Communication among devices in machine-to-machine (M2M) and Internet-of-Things (IoT) systems are an essential feature for these systems. In many M2M and IoT systems, network topology of the devices change from time to time, and network interface on devices vary from one deployment site to another one. To support dynamic connectivity and heterogeneous network protocols in the system, Multiple Protocol Network Transport (MPTN) gateway is designed to be a distributed messaging gateway to enable messaging among multiple networks. To leverage the routing capability in existing network protocols, MPTN gateway converts an end-to-end message request to a multiple segment message based on network topology. A meta-routing protocol is designed and implemented in MPTN gateway to support distributed and dynamic routing for heterogeneous networks in M2M and IoT systems. Performance evaluation shows that the proposed approach can effectively manage the routing table in distributed manner, and enable communication among multiple network protocols in M2M and IoT systems.

## Acknowledgment

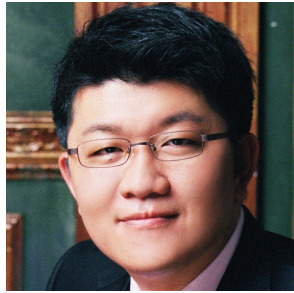
This work was supported in part by the Ministry of Science and Technology, National Taiwan University, and Intel Corporation under Grants MOST 103-2911-I-002-001, NTU-ICRP-104R7501, and NTU-ICRP-104R7501-1.

## 7. REFERENCES

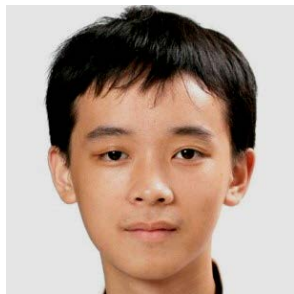
- [1] Lora alliance - wide area networks for iot. Last accessed on May 15th, 2015.
- [2] Wi-fi alliance. Last accessed on January 17th, 2015.
- [3] Z-wave technology. Last accessed on January 17th, 2015.

- [4] D. Gasco'n. Security in 802.15.4 and ZigBee networks. Last accessed on January 17th, 2015.
- [5] C. W. Liang, J. Y.-J. Hsu, and K.-J. Lin. Auction-Based Resource Access Protocols in IoT Service Systems. In *IEEE International Workshop on Internet of Things Services*, Matsue, Japan, 2014.
- [6] C. Perkins and E. Royer. Ad-hoc on-demand distance vector routing. In *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*, pages 90–100, 1999.
- [7] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers. In *SIGCOMM '94 Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, 1994.
- [8] C.-S. Shih, C.-T. Chou, K.-J. Lin, B.-L. Tsai, C.-H. Lee, D. Cheng, and J.-J. Chou. Out-of-box device management for large scale cyber-physical systems. In *the IEEE International Conference on Cyber-Physical-Social Computing*, Taipei, Taiwan, September 2014. IEEE Xplore.
- [9] C.-S. Shih, K.-J. Lin, J.-J. Chou, and C.-C. Chuang. Autonomous Service Management for Location and Context Aware Service. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 246–251. IEEE, November 2014.
- [10] P. H. Su, C.-S. Shih, J. Y.-J. Hsu, K.-J. Lin, and Y.-C. Wang. Decentralized fault tolerance mechanism for intelligent IoT/M2M middleware. In *2014 IEEE World Forum on Internet of Things (WF-IoT)*, pages 45–50. IEEE, March 2014.
- [11] C.-L. Wu, C.-W. You, C.-Y. Chen, C.-C. Chuang, and T.-C. Chiang. Exploring the Collaborative Context Reasoning in IoT Based Intelligent Care Services. In *2014 IEEE 7th International Conference on Service-Oriented Computing and Applications*, pages 241–245. IEEE, Nov. 2014.
- [12] W. Yoon and N. Vaidya. Routing exploiting multiple heterogeneous wireless interfaces: A tcp performance study. *Computer Communicatoin*s, 33(1):23–34, January 2010.

## ABOUT THE AUTHORS:



Dr. Chi-Sheng Shih joined the Department of Computer Science and Information Engineering at National Taiwan University in Feb. 1, 2004. He currently serves as an Associate Professor. He received the B.S. in Engineering Science and M.S. in Computer Science from National Cheng Kung University in 1993 and 1995, respectively. In 2003, he received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign. His main research interests include embedded systems, hardware/software co-design, real-time systems, and database systems. Specifically, his main research interests focus on real-time operating systems, real-time scheduling theory, embedded software, and software/hardware co-design for system-on-a-chip. His research results won several awards including the Best Paper Award, 2011 ACM Research in Applied Computation Symposium (RACS 2011), the Best Paper Award, IEEE RTCSA 2005, and the Best Student Paper Award, IEEE RTSS 2004.



Guan-Fan Wu received his B.S. degree from the Department of Applied Electronics Technology, National Taiwan Normal University, Taipei, Taiwan, in 2011. In the B.S. degree, his researches focus on system-on-chip (SoC) design and flexible electrophoretic display. Later, Guan-Fan Wu received his M.S. degree from the Department of Computer Science, National Taiwan University, Taipei, Taiwan, in 2013. His researches focus on embedded system and wireless routing. Currently, he is a system software engineer Nvidia to develop the display driver.

# Dynamic System Modeling of Evolutionary Algorithms

Gustav Šourek  
Czech Technical University  
Prague, Czech republic  
souregus@fel.cvut.cz

Petr Pošík  
Czech Technical University  
Prague, Czech republic  
petr.posik@fel.cvut.cz

## ABSTRACT

Evolutionary algorithms are population-based, metaheuristic, black-box optimization techniques from the wider family of evolutionary computation. Optimization algorithms within this family are often based on similar principles and routines inspired by biological evolution. Due to their robustness, the scope of their application is broad and varies from physical engineering to software design problems. Despite sharing similar principles based in common biological inspiration, these algorithms themselves are typically viewed as black-box program routines by the end user, without a deeper insight into the underlying optimization process. We believe that shedding some light into the underlying routines of evolutionary computation algorithms can make them more accessible to wider engineering public.

In this paper, we formulate the evolutionary optimization process as a dynamic system simulation, and provide means to prototype evolutionary optimization routines in a visually comprehensible framework. The framework enables engineers to follow the same dynamic system modeling paradigm, they typically use for representation of their optimization problems, to also create the desired evolutionary optimizers themselves. Instantiation of the framework in a Matlab-Simulink library practically results in graphical programming of evolutionary optimizers based on data-flow principles used for dynamic system modeling within the Simulink environment. We illustrate the efficiency of visual representation in clarifying the underlying concepts on executable flow-charts of respective evolutionary optimizers and demonstrate features and potential of the framework on selected engineering benchmark applications.

## CCS Concepts

•**Mathematics of computing** → **Evolutionary algorithms**; •**Software and its engineering** → **Data flow languages**; **Flowcharts**; *Data flow architectures*; *Visual languages*;

## Keywords

Optimization, Matlab Simulink, Evolutionary Algorithms, Dynamic systems, Data Flow, Visual Programming

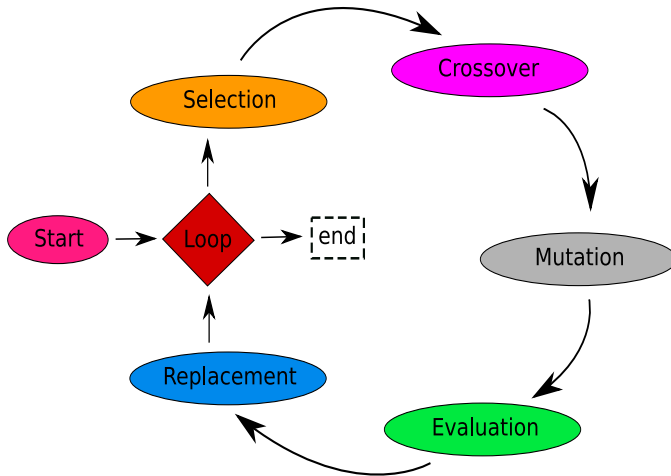
Copyright is held by the authors. This work is based on an earlier work: RACS'15 Proceedings of the 2015 ACM Research in Adaptive and Convergent Systems, Copyright 2015 ACM 978-1-4503-3738-0. <http://dx.doi.org/10.1145/2811411.2811517>

## 1. INTRODUCTION

Visual representation of information, allowing to quickly communicate and share ideas, forms an important part of scientific and engineering progress, with applications varying from physics to software. Visualization has especially proven valuable in fields where sharing and quick communication of ideas is important. Utilizing schemas and charts we exploit natural way of perceiving information in humans [21]. These underlying principles of our perception have especially been exploited in educational praxis, where quick adoption of concepts is essential. In computer science for instance, flowcharts showing operations that take place inside of an algorithm are often used to demonstrate students the function thereof. In the same manner, physical system models are often presented using schemas decomposed into elementary concepts in mechanical, civil or electrical engineering fields [18], with dynamic system modeling [23] as a prominent formalism used to describe affiliated tasks.

Traditionally, wide variety of engineering tasks from civil to control engineering can be formulated in the form of an optimization problem. This formulation naturally covers situations when we try to, e.g., minimize time, space or resources, or maximize utilization and speed, respectively. Due to their robustness, evolutionary algorithms [14] and other optimizers from the evolutionary computation family [2] have proven useful in finding high quality solutions when complex engineering systems are in scope of optimization [9]. Whereas there are many frameworks enabling engineers to employ variety of (evolutionary) optimization tools to tackle the problems in scope, these tools are typically strictly decoupled from formulation of the problem itself. This means that the functioning of the optimizer is black-box w.r.t. to the user and the system being optimized (and vice versa). The subject of this work is to overcome this limitation and merge the problem formulation with respective optimization design under a common paradigm of dynamic system modeling.

Although the visual representation has proved to be a powerful tool for rapid prototyping of dynamic system models, its practical use in computer science was traditionally limited to conceptualization only, clearly separated from the implementation phase. We aim to pass this barrier with the paradigm of graphical programming [28] to let engineers design respective evolutionary optimization algorithms in the same, natural and visually comprehensible manner. Following this paradigm we have created *VisualEA* library [32]



**Figure 1: Visualization of the evolutionary optimization algorithm cycle**

for *Matlab Simulink* [24], a graphical programming language widely adopted in variety of engineering fields, formalizing the decomposition of complex systems into atomic elements, defined by a rich set of libraries for various engineering domains. The VisualEA toolbox extends the functionality of Simulink with evolutionary and other iterative optimization algorithms from the evolutionary computation family. Although the original goal of the library was to serve as an education tool for evolutionary algorithms [31], utilizing the correspondence with visual charts that are often used in explaining the functionality of iterative optimizers, its potential has reached further. In this paper we introduce the idea of dynamic system modeling of evolutionary algorithms, present the optimization routines in scope, illustrate their corresponding visual representations, outline an architecture of the library, and demonstrate its potential on selected engineering optimization benchmarks.

## 2. EVOLUTIONARY OPTIMIZATION

Optimization is a prominent formalism used for a wide variety of tasks across different domains. Optimization can be seen as a selection of the best element from a set of available alternatives, with regard to some given criteria and a loss (fitness) function. In the engineering fields, where complex non-linear systems are often in scope of optimization, evolutionary computation methods [1] proved as a valid optimization technique for finding globally optimal or near-optimal solutions [9].

Evolutionary Algorithms (EA) are the most prominent examples from the evolutionary computation family of optimizers. EA is a generic population-based meta-heuristic optimization technique. This means that, generally speaking, it belongs to the family of “trial and error” problem solvers, iteratively improving a whole set of solutions, rather than just a single point in the search space, in a stochastic manner. This is an important factor for robustness of the optimization within the complex systems area. From computer science point of view, evolutionary computation is viewed as a subset of computational and artificial intelligence [12].

---

### Algorithm 1 Evolutionary Algorithm Routine

---

```

1  $X^{(0)} \leftarrow initialize()$ 
2  $f^{(0)} \leftarrow evaluate(X^{(0)})$ 
3  $g \leftarrow 0$ 
4 while ( $\neg terminalCondition$ ) do
5    $X_{par} \leftarrow select(X^{(g)}, f^{(g)})$ 
6    $X_{off} \leftarrow crossover(X_{par})$ 
7    $X_{off} \leftarrow mutate(X_{off})$ 
8    $f_{off} \leftarrow evaluate(X_{off})$ 
9    $[X^{(g+1)}, f^{(g+1)}] \leftarrow Draw(X^{(g)}, X_{off}, f^{(g)}, f_{off})$ 
10   $g \leftarrow g + 1$ 
11
12 return  $bestOf(X^{(g)}, f^{(g)})$  ▷ best found solution

```

---

In the process of evolutionary computation, there are generally two main forces that form the basis of optimization systems. Whereas recombination forces the necessary diversity and thereby facilitates novelty, selection acts as a force increasing the quality. The stochastic character then becomes apparent in both the recombination phase, randomly changing pieces of candidate solutions, and the selection phase, where the candidates are chosen with increasing probability based on their quality [1].

### 2.1 Family of Algorithms

The most prominent example of evolutionary computation are Evolutionary Algorithms (EA), using mechanisms inspired by biological evolution, such as reproduction, mutation, crossover, and selection, where candidate solutions to the optimization problem play the role of individuals in a population, and a fitness function, representing a generic loss function to be minimized, determines the quality of the individuals. Rather than enumerating specific algorithms, the evolutionary computation can better be described as a family of algorithms following similar optimization routines, where the evolution of the population takes place with a repeated application of the evolutionary operators, such as visualized for EA in Figure 1 and described in Algorithm 1.

In the category of evolutionary computing there are, together with EA, many other meta-heuristic population based optimization approaches inspired by biology and evolution, such as ant colonies [7], *particle swarm optimization* [4], and genetic programming [3]. Other more loosely inspired approaches, focused more on the underlying stochastic and mathematical properties, include Evolution Strategies such as covariance matrix adaptation [17] and *estimation of distribution* [22] algorithms. These methods differ in one or more aspects from the original EA routine, such as omitting the recombination phase, or by using some specialized operators, but generally follow a similar stochastic, population based, iterative routine as depicted in Figure 1, and we will introduce some representatives under the graphical programming paradigm in Section 4.1.

## 2.2 Evolution as a Dynamic System

A dynamic system is a concept from mathematics where a fixed rule set describes how a point  $s$  in a state space  $S$  depends on time  $t \in T$ . These rules are often formally viewed as an *evolution function*  $\phi \in \Phi$ , endowing the state space  $S$  by mapping each point  $s \in S$  back into  $S$  for any given time  $t \in T$ . Depending on the representation of time  $T$ , dynamic systems can be further divided into continuous and discrete, where the corresponding evolution functions  $\Phi$  are typically viewed as solutions of differential ( $\dot{x} \leftarrow \phi(x)$ ), or difference ( $x_{t+1} \leftarrow \phi(x_t)$ ) equations, respectively, with the order of the equation corresponding to the order of the modeled system [29].

There are many works utilizing evolutionary optimizers for the design and parameter tuning of dynamic systems [10, 19], prominent e.g. in the control system theory, however considering evolutionary optimizers themselves under the dynamic system paradigm, together with the system they optimize, has not yet been considered a subject of scientific inquiry. Although this view of evolutionary optimizers is not common within the community, probably for the fact that it does not offer any special advantages in the traditional optimization sense, the formulation of evolutionary optimization as a dynamic system should not cause any confusion as it is consistent and also unifies the formulation of the both the optimizer and the system in scope, which might be beneficial simplification for visual comprehensibility (Section 3.1).

Referring to the notation used in Algorithm 1, we set up a formalism where the state  $s \in S$  of the dynamic EA system (algorithm) will be given by a current population matrix  $s \leftarrow X^{(t)}$ ,  $X \in \mathbb{R}^{m \times n}$ ,  $t \in T \subset \mathbb{Z}$ , and the evolution function  $\phi \in \Phi$  will be composed of sequential aggregate of the EA operators (selection, recombination,...), operating on the actual state matrix  $X^{(t)}$  as

$$X^{(t)} = \left( \text{replace} \left( \text{mutate} \left( \text{crossover} \left( \text{select}(X^{(t-1)}) \right) \right) \right) \right)$$

This clarifies the view of the standard evolutionary routine from Algorithm 1 as a first order discrete dynamic system with evolution function  $\phi: \mathbb{R}^{m \times n} \mapsto \mathbb{R}^{m \times n}$ , i.e. formally

$$X^{(t)} \leftarrow \phi(X^{(t-1)})$$

Embracing this idea, we will be able to demonstrate that the EA routine itself can effectively be modeled as a discrete dynamic system of the first order and by those means efficiently introduced into the dynamic system model simulation environment of Simulink.

## 3. FRAMEWORK ARCHITECTURE

General idea of the framework is to introduce the process of evolutionary computation under a comprehensible visual paradigm used for dynamic system modeling. Towards that purpose there are several graphical programming environments that might be considered, but as already suggested, we decided to utilize the graphical data-flow framework of

Simulink, a popular environment primarily designed for dynamic system modeling which enables us to fully benefit from the dynamic system formulation of evolutionary computation. Moreover, Simulink is probably the most widespread modeling environment within the engineering community [20], especially considering the integration with Matlab.

In this chapter we introduce the overall framework of graphical programming of evolutionary algorithms via dynamic system modeling in Simulink. We introduce general ideas of graphical programming, and proceed from classical flowcharts to Simulink's diagrams for dynamic system modeling. We then describe the underlying data-flow principles and interfaces of the used blocks to conclude with the final architecture of the resulting library.

### 3.1 Graphical Programming

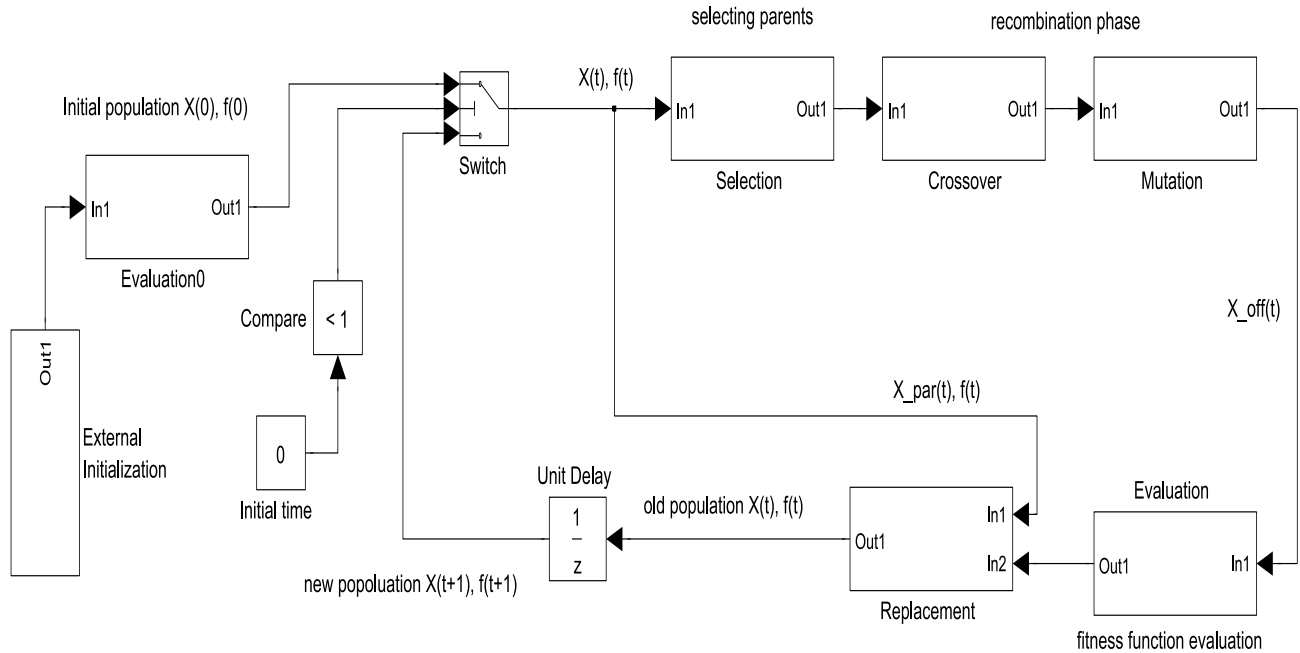
Graphical (visual) programming [28] generally tries to integrate the phases of draft visual design and rapid development of algorithms. It is becoming popular in several engineering and software related fields such as design, modeling, and data processing, especially for its natural display of common concepts for people with varying (non-programming) background. Graphical programming languages generally let users create programs by manipulating program elements graphically rather than by specifying them textually, utilizing visual expressions, spatial arrangements and various graphical symbols.

Graphical programming environments provide symbolic elements which can be manipulated by users in an interactive way according to a specific spatial grammar for program construction. These can be further divided into icon-based languages, form-based languages, and diagram languages, according to the type of visual expressions being used [25]. In design of visual programs, different shapes, colors, and spatial arrangements of operations are typically used to guide users in understanding of program's concepts. One of the overall most common visual design paradigms is the idea of "boxes and arrows" [33], where specific subparts and elements of a system can be hierarchically embedded within boxes (or other shapes), whereas the arrows correspond to information exchange, most commonly representing either control or data flow within the system (e.g., Figure 1).

In graphical programming in general, the boxes and lines can be used to describe many models of computation. One typical example is a flow chart, which also consists of blocks and lines, however, a classic flowchart paradigm is limited for our cause as one cannot describe general dynamic systems using regular flow chart semantics. To overcome the limitation of basic flow charts, standardly used in graphical programming languages, we reach to Simulink's *time-based block diagrams*. These block diagrams differ from other forms of "boxes and arrows" charts in that they explicitly describe dynamics of the modeled systems.

### 3.2 Dynamic System Modeling

Modeling dynamic systems in Simulink follows its established block diagram semantics, where the classic block diagram model of a dynamic system graphically consists of



**Figure 2: EA routine drafted as a diagram of a dynamic system in raw Simulink**

blocks and lines (signals) [24], just as in the common graphical programming paradigm. However, these block diagram models are traditionally derived from engineering areas such as feedback control theory. A block within a block diagram defines a dynamic system by itself and the relationships between each elementary dynamic system in a block diagram are illustrated by the use of signal lines connecting the blocks. These blocks and lines in a block diagram collectively describe the overall dynamic system. Simulink extends this semantics with virtual blocks for organizational convenience to improve the overall readability, which is beneficial for presenting complex evolutionary systems.

In contrast to regular flow charts, Simulink block diagrams define time-based relationships between signals and state variables (see formalism in Section 2). The solution of a block diagram is obtained by evaluating these relationships over the time of simulation. Each evaluation of these relationships is referred to as a time step [24], which in our case is the transfer from one generation of solutions to the next. Signals represent quantities that change over time and are defined for all points in time between the block diagram's simulation start and stop time. These signals in our case are matrices of population genomes and vectors of fitness evaluations (see Section 3.3 for closer description). The relationships between input signals, state variables and output signals are defined by a set of equations or methods represented by blocks. These block methods in our case are the evolutionary operators that the optimization routines consist of (see Figure 1). Inherent in the definition of these operators is the notion of parameters, which are the typically static coefficients found within respective methods (see Section 4.2 for extension). The block diagram model is then a subject to simulation carried out by one of Simulink's solv-

ers. In our case of discrete dynamic evolutionary systems, the choice is discrete fixed-time step simulation solver, since each population of solutions is generated in that fashion.

With the block diagram formalism for dynamic system simulation being clarified, we may now introduce the idea of modeling an evolutionary algorithm within the framework of Simulink by a schema depicted in Figure 2. In the schema, the EA operators, originally introduced in Figure 1, are represented as separate subsystems connected through the signal link, carrying generally the state  $s$  of the system, which, in our case, is the population matrix  $X \in \mathbb{R}^{m \times n}$  (for more details see Section 3.3). Importantly, the single *unit delay* block  $\frac{1}{z}$  denotes we work with a dynamic system of the first order and implies the invocation of the appropriate simulation solver.

### 3.3 Data-flow Principles

A data-flow programming is a modern approach in which a stream of data is passed from instruction to instruction for processing. This stands in contrast to a typically used control-flow principle, upon which the majority of main stream languages is based to date (e.g., Java, C, Python). Data-flow programming principles are beneficial for our cause, since they naturally reflect the features of both dynamic system modeling and graphical programming. The data-flow programming languages, modeling programs as directed graphs of the data flowing between operations, often form a perfect interface for visual programming to implement the desired principles. Data-flow based visual programming languages are then particularly popular for their natural grasp of parallelization [8].

In the Simulink environment, these principles of data-flow backed visual programming allow for the idea of boxes and arrows to be naturally transformed into engineering schemas, where the boxes generally represent subsystems and the arrows denote the data-flow between them (as discussed in Section 3.2). From the schema introduced in Figure 2, we can deduce that the intended block-set library should consist of hierarchical categories of blocks encompassing the individual operators from which the particular instantiations of various optimizers can be assembled, i.e., initialization, recombination, mutation, evaluation, replacement, and auxiliary control blocks. To conclude this specification, we provide the necessary data-flow interfaces for these categories in Table 1.

**Table 1: Specification of data-flow interfaces of basic EA operators transforming state of the model of evolutionary algorithm carried by signal links**

Type of block	In
	Out
Initialization	
	Initial pop. $X^{(0)}$
Fitness function	Population $X^{(t)}$
	Evaluated pop. $\{X^{(t)}; f^{(t)}\}$
Selection	Evaluated pop. $\{X^{(t)}; f^{(t)}\}$
	Population $X_{par}^{(t)}$
Recombination	Population $X_{par}^{(t)}$
	Population $X_{off}^{(t)}$
Replacement strategy	$\{(X_{off}^{(t)}; f_{off}^{(t)}) \cup (X^{(t)}; f^{(t)})\}$
	Eval. new pop. $(X^{(t)}; f^{(t)})$
Iterator	$(X^{(t)}; f^{(t)})$
	$(X^{(t+1)}; f^{(t+1)})$

### 3.4 A Simulink's Library

Practical contribution of the introduced framework is extending Simulink with a block-set library of the operators from evolutionary computation, for which we have established a simple formalism of evolutionary optimization as a discrete dynamic system in Section 2.2, which was an important step in clarification of the software architecture of the VisualEA library. Firstly, unlike the natural perspective following object-modeling approach, from the dynamic system perspective it is clear that units within the population should be considered jointly by the means of a population matrix determining the state of the algorithm and the signal carried between subsystems. This choice of representation enables greater flexibility, further extendability of the library, and also naturally exploits matrix transformation capabilities of Matlab-Simulink. Secondly, the first order discrete system point of view clarifies the state-space split

of the optimization routine in terms of unit delay blocks  $\frac{1}{z}$ . This formulation also forces Simulink to use the fixed-step size discrete iterative simulation mode, rather than calling typical “algebraic loop” solvers.

The Simulink environment provides a graphical editor that allows to create and connect instances of block types selected from block libraries via a library browser. Libraries of blocks are provided representing elementary systems that can be used as building blocks. Apart from built-in blocks Simulink enables to provide custom block types and use the editor to create instances of these in a diagram. There are several ways to bring this custom functionality into Simulink, either graphically or programmatically. The most natural and convenient way is to create the functionality graphically while building on top of the existing blocks. Unfortunately these do not provide sufficient functionality for our cause. Second option is to integrate custom code into the system, for which there are following options.

- Function (basic mathematical expressions)
- Matlab function (all expressions from Matlab)
- Embedded Matlab function (static subset of Matlab)
- S-function (Simulink-specific language for blocks)

The VisualEA block-set was implemented almost solely using *embedded Matlab*, a subset of the Matlab language putting a number of static restrictions on the language to enable a direct generation of C code from Matlab algorithms. This option retains most of the expressiveness of Matlab and offers a considerably efficient way to implement the required functional blocks, where each one implements a common interface for the given data-flow category (Table 1) of operators. Using embedded Matlab for implementation of VisualEA we do not trade-off comprehensibility for performance, as is usually the case with graphical programming, but leverage both at the same time (as compared to a regular Matlab implementation).

From the specifications in Table 1, we can notice that the signal carried between the blocks might actually be of different types, including a population matrix  $X \in \mathbb{R}^{m \times n}$  and a vector of individual fitness values  $f \in \mathbb{R}^m$ , which we omitted in the previous formalism (Section 2.2) for simplicity (the signal format may further differ for different variants of evolutionary optimizers introduced in Section 4.1). The fitness values are created within the fitness function block, encompassing the generator  $g$  of evaluations  $f \leftarrow g(x)$  for the candidate solutions  $x \in \text{rows}(X)$  of EA, which can be represented by another (dynamic) system sharing the given data-flow interface. We will provide examples of custom fitness systems in Section 5.

This completes the key idea and the specification of the core building blocks of VisualEA, displayed in Figure 4. The concept and structure of the models rises naturally from the draft design outlined in Figure 2. Utilizing the library [32], we can now translate the draft Simulink schema into *executable model*, a particular instantiation of which is displayed in Figure 3. This schema can now be simply *run* to perform

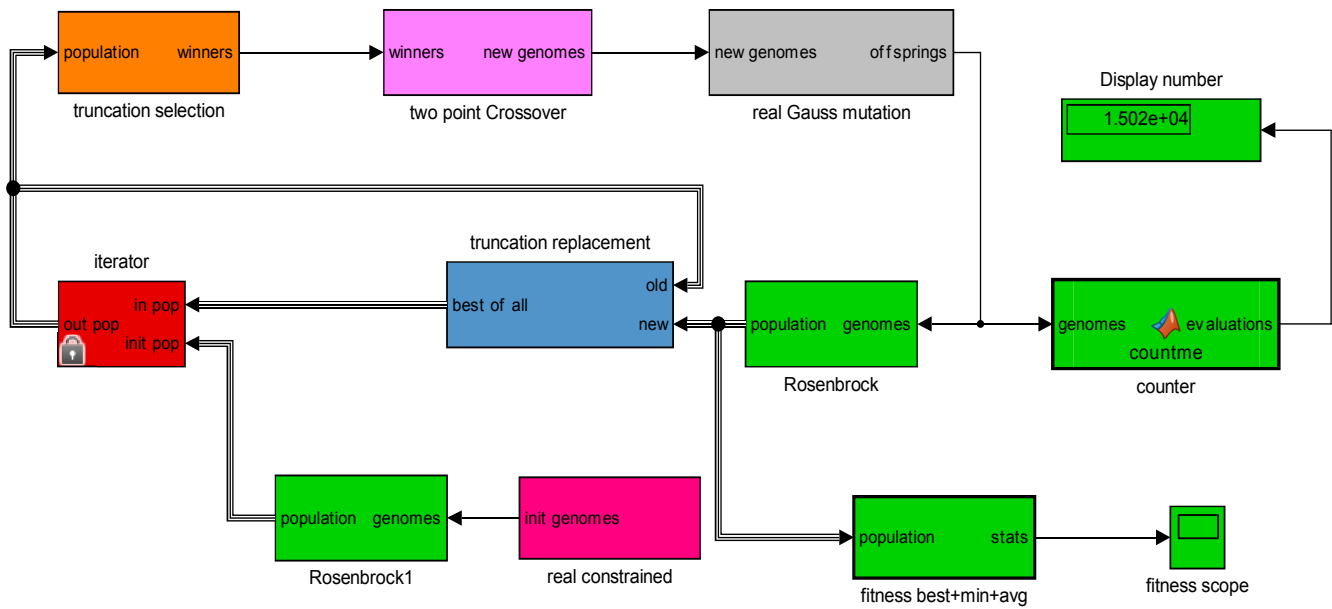


Figure 3: A basic EA model with selected types of evolutionary operators from VisualEA library (Fig. 4)

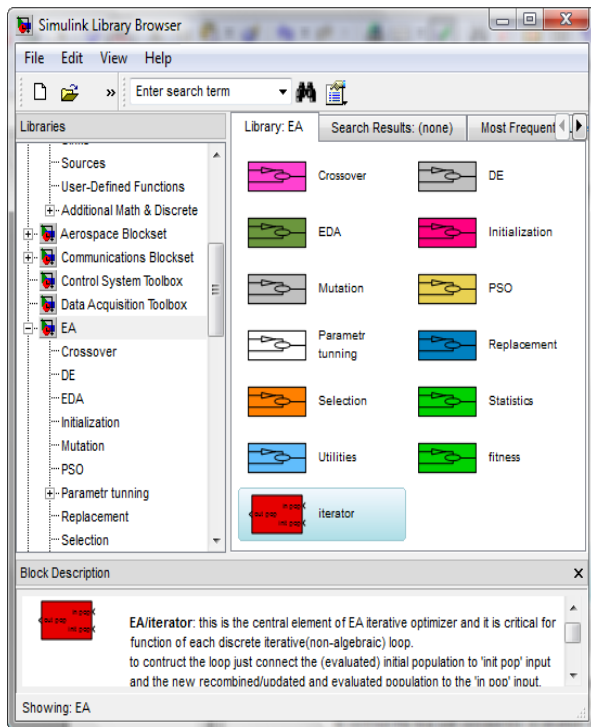


Figure 4: Simulink's interface of VisualEA library

the optimization routine on a given input system embedded within the *user-defined* fitness block (see Section 5 for fitness examples). We can see that the actual model does not principally differ from the draft model we outlined in raw Simulink, and should thus be comprehensible from the pure engineering point of view. Briefly speaking, we only

covered the unit delay and switch operators within one *iterator* block for better readability. The colors for block-sets from the library were selected to denote operators from the same category (Table 1), so as to guide user in understanding and intuitive assembling of optimization loops (e.g., green color denotes all fitness-related operations), in correspondence with the original visualization in Figure 1. The final model is thus also intelligible, without any classical programming knowledge, as a pure “boxes and arrows” concept under the general visual design paradigm.

## 4. FRAMEWORK FEATURES

The introduced VisualEA library implements functionality of evolutionary optimization algorithms via simulation of their respective dynamic system models. Although this rather unorthodox architecture might seem intriguing at the beginning, as compared to e.g. standard (visual) programming methods, it actually provides a number of interesting features that are not typical in programming frameworks but come along naturally with dynamic system modeling. In this chapter we discuss two selected features the approach provides. Firstly the flexibility of creating various optimization models we aimed at, which is based on their decomposition into operators represented by atomic blocks of the library. Secondly, we describe the merit of online tuning of meta-parameters of the evolutionary models during simulation (without recompiling).

### 4.1 Different Evolutionary Optimizers

The introduced concept of VisualEA is not bound to a single EA implementation, and rather offers a flexible framework to form the whole family of evolutionary optimizers, some of which we introduced in Section 2. Some variants can be formed with just a slight modification of the model from

Figure 3, such as the generational model of EA, where units live just for the time of one generation, which can be simply realized by omitting the replacement strategy block. Also evolutionary strategies, using various generation or steady-state models utilizing selection at the end of evolutionary cycle, can be simulated as such through the replacement strategy block.

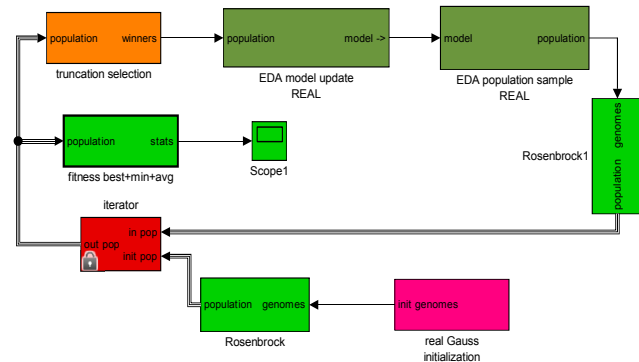


Figure 5: Estimation Of Distribution optimizer modeled with VisualEA

However some of the introduced optimizers require further modifications to be formed. As an example we provide the estimation of distribution (EDA) family, utilizing probabilistic model learning and sampling to produce new generation instead of the recombination operators, a model thereof is displayed in Figure 5. Another popular example might be the particle swarm optimization group, utilizing generational model with unique recombination operators based on information about actual and global best found positions of particles, a model thereof is provided in Figure 6.

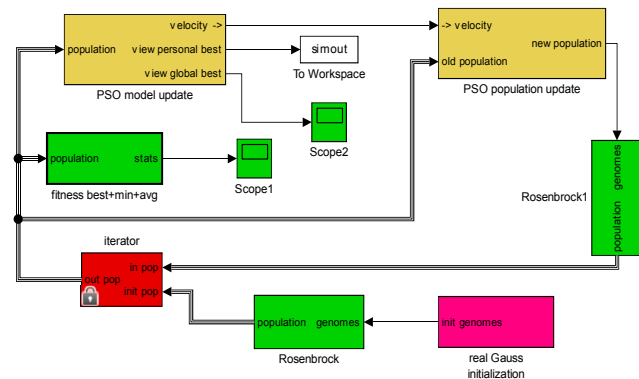


Figure 6: Particle Swarm Optimization algorithm modeled with VisualEA

## 4.2 Real-time Parameter Tuning

Parameter tuning of evolutionary algorithm routines is an inherently tricky part of most of the optimizers [11] and finding appropriate parameter values for evolutionary algorithms is a persisting grand challenge of evolutionary computing. While all researchers acknowledge that a proper choice of parameter values is essential for desired performance of their algorithms, the tuning process is still mostly

driven by conventions (e.g. that a mutation rate should be low), ad-hoc choices, repeated attempts and runs of the algorithm to understand the behavior of the, more or less black box, evolutionary optimization system in scope. While there are strategies for a more sound parameter tuning of EAs, e.g. by grid searching or embedding within another meta-EA [30], these typically ignore the time-complexity of evolutionary computation which is often extreme, and thus the manual parameter tuning remains a usual practice.

With the visual paradigm we can provide insights not only into the algorithm’s principal architecture, but the dynamic system simulation approach provides insight into the optimization process itself, too. This can be especially useful for the discussed meta-parameter tuning. Similarly to any dynamic system, we may observe various characteristics of the optimizer during the run, such as the typically viewed progress of fitness. But on top of a mere observation we can also provide online feedback into the model, which is one of the most interesting features of the approach. This functionality is based on *tunable parameters* in Simulink. A tunable parameter is a parameter whose value can be changed during the simulation without recompiling the model.

For instance, we might want to observe progress of optimizer’s performance, such as the global minimum, current( $t$ ) minimum and average( $t$ ) fitness, provided by VisualEA within a regular Simulink scope as displayed in Figure 7, to interfere with the model simulation by interactively changing the recombination, e.g., increasing the mutation rate when the exploration seems too slow or once we get stuck in a poor local optima. In this manner we might interactively explore the dependence between different meta-parameters, while observing the fitness dynamics to find optimal parameter settings. In this manner we may efficiently incorporate overall conventions with particular expert knowledge of the algorithm through a reasonable amount of interactive manual experimentation.

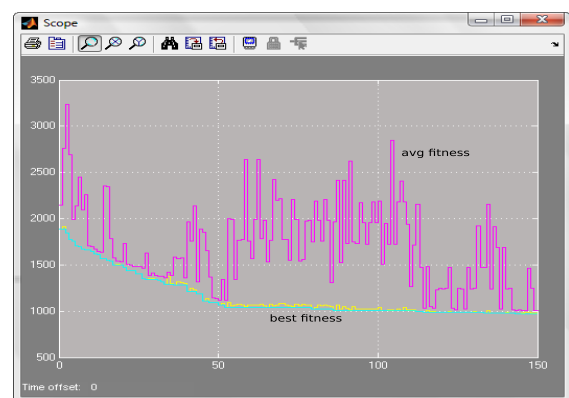


Figure 7: A regular Simulink scope displaying the fitness characteristics of the evolutionary optimization run. A result of real-time parameter tuning is demonstrated on the average fitness progress (magenta line), by manually increasing the mutation rate at simulation time  $t = 50$

## 5. BENCHMARK APPLICATIONS

The framework of VisualEA has been successfully utilized in several engineering benchmarks and applications. We will provide examples of two of those for a brief demonstration. The way to incorporate a (dynamic) system as a subject for optimization is to include it within the fitness function block (Figure 3). This naturally follows from the standard EA schema (Algorithm 1). The library itself provides a number of benchmark fitness functions for testing, e.g. the Rosenbrock function used in some of the previous figures. Apart from predefined functions (or systems), there is an option of user-defined fitness function. This allows to incorporate not only any system defined by a Simulink schema, but also to call any external system defined by a Matlab function in its full expressiveness.

In the first example we leave the scope of sample fitness functions and demonstrate the ability to pass a non-trivial optimization benchmark, with potential applications for manufacturing, on the problem commonly known as “packing equal circles” (PECs) into a unit square [15]. In this problem we try to find the maximal diameter of  $n$  equal non-overlapping circles within a square of unit size. This notion can be reformulated by bounding just the circle centers and recalculate the diameter afterwards. Under this formulation we are thus maximizing the minimal pair-wise distance of  $n$  points inside a unit square, which can be formulated as follows.

$$\operatorname{argmax}_{\vec{x}, \vec{y}} (d) \quad s.t. \quad (1)$$

$$(x_i - x_j)^2 + (y_i - y_j)^2 \geq d^2 \quad (2)$$

$$\forall i < j \leq n; \quad ; \quad x_i, y_i \in (0, 1) \quad (3)$$

The PECs problem has been solved mathematically for  $n = 1..9, 16, 25, 36$  instances and other small or symmetric packings using branch and bound algorithms. However, PECs problem is commonly considered to be NP-complete for general  $n$ , and bigger instances thus remain a subject for meta-heuristic optimization, making the approach of evolutionary optimization appropriate. With some modifications of the original fitness representation, we successfully utilized VisualEA to solve this problem on several difficult instances from  $n = 7..23$ , with results illustrated in Figure 8 and 9 for  $n = 7$  and  $n = 23$ , respectively.

In the second example, we demonstrate the potential of VisualEA to optimize significantly complex engineering systems. The system in scope represents a generic bridge construction (inspired by the game Bridge Builder). In the given optimization setting, we try to find the best stable bridge structure consisting of the least possible number of girders provided their maximum tension limits. To bound the originally infinite solution space of all possible structures, every constellation of girders is laid out onto a triangular grid represented by a planar graph, where the girders correspond to edges and their joints to nodes, respectively. Based on this triangular grid we formalize the structure representation into a finite set of elements, expressing the horizontal and vertical stability of the overall structure as

$$\forall j \in \mathcal{J} : \sum_{g \in \text{Girders}(j)} f(g) \cos(\theta(g, j)) = 0 \quad (4)$$

$$\sum_{g \in \text{Girders}(j)} f(g) \sin(\theta(g, j)) = 0 \quad (5)$$

$$\forall g \in \text{Girders}(j) \exists z : e(z, j) \vee e(j, z) \in \text{Grid} \quad (6)$$

where  $f(g)$  are tension forces applied to each girder  $g$  corresponding to some edge  $e(a, b)$  in the original *Grid* graph, grid joints  $j \in \mathcal{J}$  corresponding to the nodes, and the function  $\theta(g, j)$  denoting the angle of a particular girder  $g$  w.r.t. grid joint  $j$ .

This problem setting generally falls into the category known as multi-disciplinary system design or simply structural design optimization. There are typically two streams of approaches depending on the complexity of particular problems. For simpler formulations, mainly the methods of linear programming and gradient based solvers are used. For more complex problems with non-linear constraints and non-convex solution space, more robust heuristic approaches are used, with evolutionary algorithms as the most prominent example [27]. To demonstrate the integration capabilities of VisualEA, we propose an approach that combines the evolutionary optimization with linear programming routines.

The encoding of a bridge construction solution genome is a matrix of girders and their joints. Thanks to the grid structure with a predefined fixed set of angles (and thus precomputed cosines and sines in Equations 4 and 5), each of the joints encodes two linear equations for horizontal and vertical equilibrium of forces applied to it. Combined with the constraints of maximal tension (and compression) limit of each girder as

$$\forall j \in \mathcal{J} : \forall g \in \text{Girders}(j) : |f(g)| \leq \text{limit}(g) \quad (7)$$

we may set up a routine for linear programming to solve the optimal tension distribution of a stable bridge structure.

Remaining is the distribution of the girders itself, which imposes an inherently non-convex character on the problem and is left to the evolutionary strategy. The evolutionary algorithm then searches the space of particular bridge architectures, which are being evaluated by the routines of linear programming for each individual solution genome.

Starting from a random architecture, we let the evolutionary algorithm converge towards a better architecture using lesser number of girders while keeping the bridge structure stable. Utilizing the linear programming routines greatly improves the convergence of the algorithm and this is where the merit of integration with Matlab becomes apparent. Utilizing Matlab within the custom fitness block, we have access to the powerful built-in linear programming functions and other features of the environment developed for variety of engineering domains. Again, we provide illustration of the results achieved with VisualEA in Figures 10 and 11 for brief illustration.

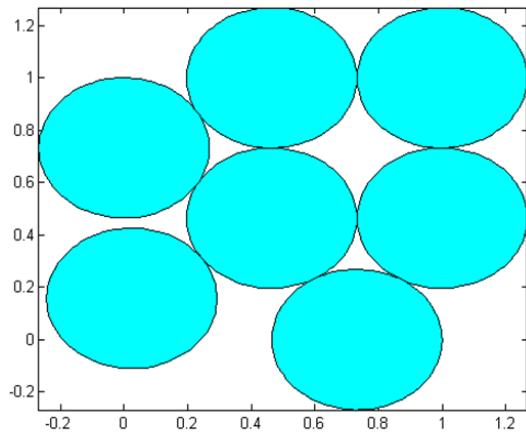


Figure 8: A small PECS problem solved with VisualEA on instance of 7 circles

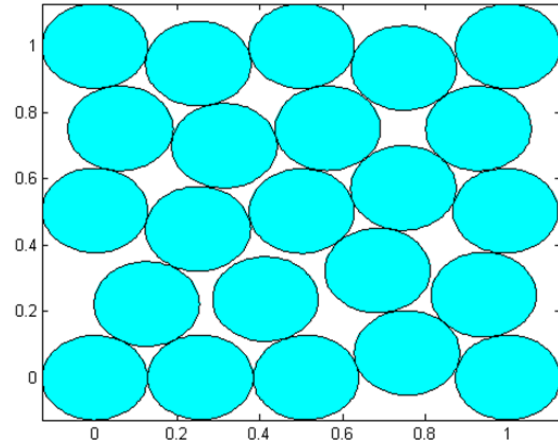


Figure 9: A bigger PECS problem solved with VisualEA on instance of 23 circles

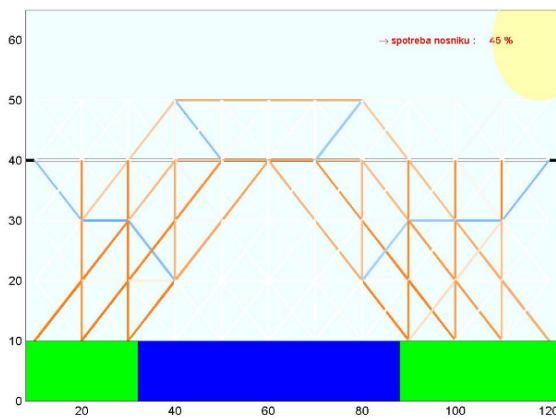


Figure 10: A small bridge construction evolved with VisualEA

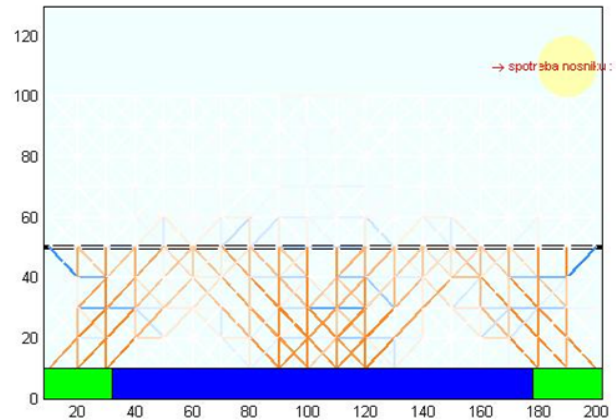


Figure 11: A bigger bridge construction evolved with VisualEA

## 6. RELATED WORK

From the most closely related works, there was an Evolutionary Algorithm Composer project [13] where some variants of evolutionary algorithms were possible to be created in a diagrammatic way to free users from programming the standard routines. This custom tool translated the graphical input into a Java program that might be run normally. While similarly to VisualEA this tool provides means for graphical design of some evolutionary algorithm routines, it differs in the paradigm of modeling evolutionary optimizers as dynamic systems, which provides us with further features of real-time simulation. An important distinction is also that in contrast to being a custom Java tool, VisualEA builds on an existing data-flow interface of Matlab-Simulink, which makes it easy to integrate with arbitrary problems and systems complying to the respective environment.

A Matlab-Simulink based approach utilizing evolutionary algorithms, directed more closely towards system simulation and modeling, was introduced in [16]. In their work the authors target the problem of parameter and structure optimization of Simulink models that may represent various dynamic systems to be optimized. The optimization method is based on a predefined system entity structure/model framework. While interesting for bringing the structural evolutionary optimization into the model management in Simulink, the work principally differs from our approach as the evolutionary computation itself is not a part of the considered model and is used purely as a preprogrammed black-box optimization technique.

The wider body of related work naturally comprises of approaches combining visual paradigms and evolutionary or genetic algorithms. In the work [6], the authors focused on

using visualization technology to guide users in validating their evolutionary solutions. The authors review existing techniques for presenting evolutionary optimization output data and introduce dimension reduction techniques for visualizing the search space in a format that is easy to understand. While they also strive to bring more light into the evolutionary computation, they focus purely on the output solutions through the population matrix visualization and not the evolutionary process itself. In a similarly directed work [34] the authors analyze techniques for visualization of the genome within population. Applying different aggregations and statistical methods they try to investigate the details of the evolutionary run through the genome analysis, reaching beyond the standard pure fitness progress visualization. Another paper [26] presents a set of standard visualization techniques for different data types and time frames of the evolutionary algorithm, according to their usefulness for real world applications. To gain the visual insights, the authors developed a graphical user interface to access the different visualization methods and styles during and after an optimization, which became a part of the Genetic and Evolutionary Algorithm Toolbox for Matlab [5]. While close in either the visualization spirit or the used technology, the distinguishing feature of our approach to all the previous works is the graphical dynamic system modeling approach to evolutionary optimization.

## 7. CONCLUSIONS

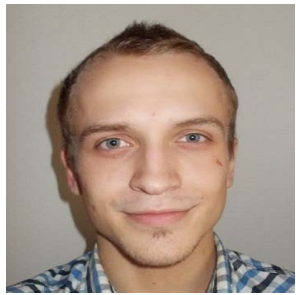
In this paper, we proved conceptual validity of understanding and modeling evolutionary algorithms as discrete dynamic systems and demonstrated potential benefits of such an approach. We have introduced a new data-flow framework for visual understanding and graphical programming of optimizers from the evolutionary computation family under the paradigm of dynamic system modeling. We presented the standard evolutionary routine within the formalism of dynamic systems and by those means introduced it into the Matlab-Simulink environment. The concept of graphical dynamic system modeling of evolutionary optimizers was then embodied within a Simulink library, which allows to create fully executable models of various optimizers following the given data-flow architecture. We demonstrated the visual comprehensibility and flexibility of the concept on several models, and outlined some interesting features of the framework. Finally, we validated the potential of the library on selected optimization benchmarks.

## 8. REFERENCES

- [1] T. Back. Evolutionary algorithms in theory and practice. 1996.
- [2] T. Back, D. B. Fogel, and Z. Michalewicz. *Handbook of evolutionary computation*. IOP Publishing Ltd., 1997.
- [3] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic programming: an introduction*, volume 1. Morgan Kaufmann San Francisco, 1998.
- [4] C. Blum and X. Li. *Swarm intelligence in optimization*. Springer, 2008.
- [5] A. Chipperfield and P. Fleming. The matlab genetic algorithm toolbox. In *Applied control techniques using MATLAB, IEEE Colloquium on*. IET, 1995.
- [6] T. D. Collins. Using software visualisation technology to help evolutionary algorithm users validate their solutions. In *ICGA*, pages 307–314. Citeseer, 1997.
- [7] A. Colorni, M. Dorigo, V. Maniezzo, et al. Distributed optimization by ant colonies. In *Proceedings of the first European conference on artificial life*, volume 142, pages 134–142. Paris, France, 1991.
- [8] P. Cox, S. Gauvin, and A. Rau-Chaplin. Adding parallelism to visual data flow programs. In *Proceedings of the 2005 ACM symposium on Software visualization - SoftVis '05*, page 135, New York, New York, USA, May 2005. ACM Press.
- [9] D. Dasgupta and Z. Michalewicz. *Evolutionary algorithms in engineering applications*. Springer Science & Business Media, 1997.
- [10] R. C. Eberhart and Y. Shi. Tracking and optimizing dynamic systems with particle swarms. In *Evolutionary Computation. Proceedings of the 2001 Congress on*, volume 1, pages 94–100. IEEE, 2001.
- [11] A. E. Eiben and S. K. Smit. Parameter tuning for configuring and analyzing evolutionary algorithms. *Swarm and Evolutionary Computation*, 2011.
- [12] D. B. Fogel. *Evolutionary computation: toward a new philosophy of machine intelligence*, volume 1. John Wiley & Sons, 2006.
- [13] R. Fröhling. Evolutionary algorithm composer - ein werkzeug zur codegenerierung multiploider evolutionärer optimierungsalgorithmen. Diplomarbeit, Universität-GH Paderborn, Fachbereich Elektrotechnik und Informationstechnik, Fachgebiet Datentechnik, November 2001.
- [14] D. E. Goldberg. Genetic and evolutionary algorithms come of age. *Communications of the ACM*, 37(3):113–119, 1994.
- [15] M. Goldberg. The packing of equal circles in a square. *Mathematics Magazine*, pages 24–30, 1970.
- [16] O. Hagendorf, T. Pawletta, and R. Larek. An approach to simulation-based parameter and structure optimization of MATLAB/Simulink models using evolutionary algorithms. *Simulation*, 89(9):1115–1127, aug 2013.
- [17] N. Hansen and A. Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *Evolutionary Computation., Proceedings of IEEE International Conference on*, pages 312–317. IEEE, 1996.
- [18] K. Henderson. Flexible Sketches and Inflexible Data Bases: Visual Communication, Conscriptio Devices, and Boundary Objects in Design Engineering. *Science, Technology & Human Values*, 16(4):448–473, 1991.
- [19] X. Hu and R. C. Eberhart. Adaptive particle swarm optimization: detection and response to dynamic systems. In *Computational Intelligence, Proceedings of the World on Congress on*, volume 2. IEEE, 2002.
- [20] S. T. Karris. *Introduction to simulink with engineering applications*. Orchard Publications, 2006.
- [21] D. Keim. Information visualization and visual data mining. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):1–8, 2002.

- [22] P. Larrañaga and J. A. Lozano. *Estimation of distribution algorithms: A new tool for evolutionary computation*, volume 2. Springer Science & Business Media, 2002.
- [23] D. Luenberger. *Introduction to dynamic systems: theory, models, and applications*. Wiley, 1979.
- [24] MathWorks. Official Mathworks site of Matlab-Simulink product.  
<http://www.mathworks.com/products/simulink/>. Accessed: 2015.
- [25] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.
- [26] H. Pohlheim. Visualization of evolutionary algorithms-set of standard techniques and multidimensional visualization. In *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 1, pages 533–540. San Francisco, CA., 1999.
- [27] T. Ray, K. Tai, and K. C. Seow. Multiobjective design optimization by an evolutionary algorithm. *Engineering Optimization*, 33(4):399–424, 2001.
- [28] N. C. Shu. *Visual programming*. Van Nostrand Reinhold New York, 1988.
- [29] K. Sims. Interactive evolution of dynamical systems. In *Toward a practice of autonomous systems: Proceedings of the first European conference on artificial life*, pages 171–178, 1992.
- [30] S. K. Smit and A. E. Eiben. Comparing parameter tuning methods for evolutionary algorithms. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 399–406. IEEE, 2009.
- [31] G. Sourek. VisualEA first introduced as an educational tool in Bachelor Thesis. <https://cyber.felk.cvut.cz/research/theses/papers/111.pdf>. Accessed: 2015.
- [32] G. Sourek. VisualEA library published on Matlab Central FileExchange. <http://www.mathworks.com/matlabcentral/fileexchange/50128-visualea...> Accessed: 2015.
- [33] B. Verheij. Argumentation support software: boxes-and-arrows and beyond. *Law, Probability and Risk*, 6(1-4):187–208, 2007.
- [34] A. S. Wu, K. De Jong, D. S. Burke, J. J. Grefenstette, C. L. Ramsey, et al. Visual analysis of evolutionary algorithms. In *Evolutionary Computation. Proceedings of the 1999 Congress on*, volume 2. IEEE, 1999.

## ABOUT THE AUTHORS:



Gustav Šourek received his Master's degree in Artificial Intelligence in 2013 at Czech Technical University in Prague. Currently, he is pursuing PhD in Artificial Intelligence and Biocybernetics at the same University, where he also works as a researcher in Intelligent Data Analysis group. His research focus is mainly on Statistical Relational Machine Learning and affiliated problems of learning with logic, graphs and neural networks. He applies his algorithms in different relational domains, e.g., computer network security.



Petr Pošík received his Diploma degree in Technical Cybernetics in 2001 and Ph.D. in Artificial Intelligence and Biocybernetics in 2007, both from the Czech Technical University in Prague, Czech Republic. From 2001 to 2004 he also worked as a statistician, analyst and lecturer for StatSoft, Czech Republic. Since 2005 he works as a researcher at the Department of Cybernetics, Czech Technical University. Being on the boundary of optimization, statistics and machine learning, his research interests are aimed at improving the characteristics of evolutionary algorithms with techniques of statistical machine learning. He has published 5 journal articles, and over 25 conference papers. Petr is also a member of the editorial board of the Evolutionary Computation Journal, MIT Press, and serves as a reviewer for several journals and conferences in the field of evolutionary-computation.

# Improving Random Write Performance in Homogeneous and Heterogeneous Erasure-Coded Drive Arrays

Nikolaus Jeremic  
Architecture of Application  
Systems Group  
University of Rostock  
Rostock, Germany  
nikolaus.jeremic@uni-  
rostock.de

Helge Parzyjegl  
Architecture of Application  
Systems Group  
University of Rostock  
Rostock, Germany  
helge.parzyjegl@uni-  
rostock.de

Gero Mühl  
Architecture of Application  
Systems Group  
University of Rostock  
Rostock, Germany  
gero.muehl@uni-  
rostock.de

## ABSTRACT

In data storage systems, drive arrays known as RAIDs are often used in order to avoid data loss and to maintain availability in the event of drive failure(s). RAID schemes define various drive array organizations (denoted as RAID levels) that can be used in arrays of hard disk drives (HDDs) and arrays of NAND flash memory solid-state drives (SSDs). For larger drive arrays, using data striping with erasure coding is appealing due to its notably higher space efficiency compared to data replication. However, the main issue of data striping with erasure coding is the performance of random writes smaller than a stripe. This problem is even aggravated if the random access performance characteristics of the deployed device type (HDD or SSD) and device model are not properly considered when choosing the data striping configuration (in particular the stripe unit size).

In this article, we provide an analytical model allowing to predict the random write throughput of homogeneous drive arrays as well as of a heterogeneous drive array with code blocks stored on the faster drives. Based on our model, we develop a method to improve the random write throughput in homogeneous drive arrays (comprising only one device type, e.g., HDDs or SSDs) by adapting the data striping configuration to the used device type and model in relation to the workload. Then, based on our previous work, we describe an organization for heterogeneous drive arrays, which is especially suitable for arrays combining HDDs with SSDs, and permits to further increase the random write throughput by storing data blocks on slower and code blocks on faster drives. Finally, we experimentally evaluate our analytical claims and show that random write throughput can indeed be notably increased in drive arrays that use data striping with erasure coding.

## CCS Concepts

•Information systems → Flash memory; Disk arrays; RAID; Storage virtualization;

Copyright is held by the authors. This work is based on an earlier work: SAC'15 Proceedings of the 2015 ACM Symposium on Applied Computing, Copyright 2015 ACM 978-1-4503-3196-8. <http://dx.doi.org/10.1145/2695664.2695696>

## Keywords

Heterogeneous data storage systems, Hard Disk Drives, Solid-State Drives, Data Striping, Erasure coding, RAID

## 1. INTRODUCTION

In data storage systems, using RAIDs is a common approach to increase performance (compared to a single drive) and reliability (to avoid data loss and to maintain data availability in case of drive failures) at the same time. From the various RAID levels known and applied in practice, those combining data striping with erasure encoding (e.g., RAID-5 and RAID-6) are appealing to reach fault tolerance especially for drive arrays comprising many drives, because they exhibit a notably higher space efficiency than RAID levels applying data replication (e.g., RAID-1 and RAID-10). However, the Achilles heel of data striping with erasure coding is the performance of random writes that are smaller than a stripe. This so-called *small write penalty* [4, Sect. 4.1] stems from the additional requests necessary to re-compute the erasure code(s) and update the accompanying code blocks for every write request. While this effect is in principle inevitable, in practice, the potential write performance of erasure-coded drive arrays often cannot be fully exploited, because the random access characteristics of the deployed device type (HDD or SSD) and its model-specific parameters are not properly considered when choosing the data striping configuration and, in particular, the stripe unit size.

This motivated us to develop a method to adapt the data striping configuration for homogeneous drive arrays to the used device type (i.e., HDD or SSD) as well as to the characteristic parameters of the used device model while considering the workload. In case of HDDs, this requires the knowledge of two parameters (the average head positioning time and the data transfer rate), while in case of SSDs, it suffices to obtain the effective flash page size. In case of HDDs, our method increases the random write throughput when the majority of requests does not exceed a certain request size threshold, which depends on the used HDD model as well as on the number of HDDs and erasure codes applied. In case of SSDs, such a restriction does not exist. The proposed method is based on an analytical model to forecast

the random write throughput of homogeneous HDD arrays and homogeneous SSD arrays that apply data striping together with erasure coding for an arbitrary number of erasure codes. This model can be used to tune the stripe unit size to the devices' characteristics in order to maximize the random write performance.

Then, we apply the developed model to a specific organization for heterogeneous drive arrays consisting of HDDs and SSDs, which we have proposed in a previous paper [12]. This drive array organization increases the random write performance by storing data blocks on HDDs, while code blocks are stored on SSDs. This scheme is motivated by the fact that code blocks are much more frequently accessed than data blocks if small random writes comprise a substantial part of the workload. This raises the possibility for an optimization by storing the code blocks on the faster drives effectively offloading the code block requests from the slower drives. Since the code blocks only require a fraction of the capacity required for storing the data blocks, the limited capacity of the faster drives is, thus, exploited more efficiently. We call this approach *Code Block Request Offloading (CBRO)*. Based on our new results for homogeneous drive arrays, we present an updated analysis of CBRO and find that CBRO is also suitable for notably larger requests.

We present a detailed experimental evaluation using Linux Software RAID that confirms our analyses and supports our claims on a significant improvement of the random write throughput in homogeneous HDD RAIDs and SSD RAIDs as well as in heterogeneous drive arrays applying CBRO.

The remainder of the article is structured as follows: In Sect. 2 we provide information on the different random access performance characteristics of HDDs and SSDs (based on NAND flash memory). Sect. 3 explains the random write penalty of erasure coding-based schemes and provides a model to predict the random write throughput in homogeneous drive arrays. Sect. 4 presents a method to improve the random write throughput in homogeneous (erasure-coded) drive arrays by taking the device type and model into account. Sect. 5 describes our approach to alleviate the random write penalty in heterogeneous drive arrays. In Sect. 6, we present an experimental evaluation based on drive arrays comprising HDDs and/or SSDs using Linux Software RAID, showing the benefits of our approach. Related work is discussed in Sect. 7. Finally, Sect. 8 concludes the article and derives further research directions.

## 2. STORAGE DEVICE PERFORMANCE

In this section, we examine the key factors that lead to fundamentally different random access performance characteristics of NAND flash-based SSDs when compared to HDDs.

### 2.1 Hard Disk Drives (HDDs)

The service time of a data request issued to an HDD corresponds primarily to the sum of the *command overhead* (time that drive electronics needs to handle a request), *seek time* (time to position the head stack over the desired track), *rotational latency* (time to wait until the start of the desired sector reaches the read/write head), and *data transfer time* (which depends on the media data rate and the host interface

data rate) (cf. [10, Sect. 19.1]). The value ranges and especially the maximum value of these service time components are specific to the particular HDD model. HDDs process data requests consecutively [23] even if multiple outstanding requests are available at a time. However, if multiple requests occur simultaneously, the seek time and rotational latency of a request can be reduced (compared to serving the requests in the order of their arrival) through request reordering [24]. For the sake of brevity, we denote the sum of command overhead, seek time, and rotational latency of a data request as *head positioning time* in the following.

In general, HDDs achieve the higher performance (i.e., higher throughput and lower latency) the lower the head positioning time of the data requests is. However, for random accesses, a significant head positioning time (whose value depends on the used HDD model) will occur for each request, thus, limiting the achievable *Input/Output Operations Per Second (IOPS)* and also imposing a lower bound on the latency. As a result, the latency as well as the amount of data transferable per unit of time depends primarily on the request size (for random accesses). Small (random) requests will result in low amount of data transferred per unit of time (and high latency), hence, representing the least efficient operating mode for HDDs. Consequently, the random access throughput in terms of *mebibytes per second (MiB/s)* increases with growing request size, until it converges to the media data rate (which depends on the used HDD model and is usually notably lower than the host interface data rate). In contrast to that, the throughput in term of IOPS remains nearly constant with growing request size, then it starts to decline, and finally it declines linearly. This is because the head positioning time dominates the time required to serve small requests, while the media data rate dominates the time needed to serve large request (cf. Fig. 5 in Sect. 6.2.1).

### 2.2 NAND Flash Memory SSDs

Contrary to HDDs, SSDs based on NAND flash memory have no mechanical components because they use semiconductor memory to store data. Another significant difference between HDDs and SSDs (based on NAND flash memory) is the inherent capability of SSDs to process multiple data requests in parallel. This capability results from the internal architecture of SSDs. Considering the system-level (hardware) architecture of SSDs, an SSD usually accommodates multiple NAND flash memory packages connected to the SSD controller through multiple independent channels (i.e., buses). At flash-level, each package comprises multiple dies composed of several planes (often two planes per die) that can be accessed simultaneously. Each plane comprises usually several thousands of flash blocks, which consist of a certain number of flash pages (a page is the read/write unit) with a net capacity of a number of kibibytes (KiB), which is usually a power of two (e.g., 2, 4, 8, or 16 KiB). Similarly to the capacity of a single flash page, a flash block contains usually a number of flash pages that is either a power of two (often 64, 128 or 256) or a sum of powers of two (e.g., 192 or 384). The flash page size as well as the number of pages per block often depend on the used flash medium type distinguished by the number of bits stored per cell (SLC: 1 bit per cell, MLC: 2 bits, TLC: 3 bits, and QLC: 4 bits). The used

flash medium type (along with the flash cell size) has a significant influence on the time needed to read or write (known as programming) a flash page, hence, also on the performance of an SSD. Which flash medium type is used depends on the considered SSD model just like the degree of hardware redundancy resulting from a particular system-level (number of independent channels and NAND flash packages) and flash-level hardware architecture (number of dies, planes, pages per flash block, and page size).

Beside model-specific architectural properties, the performance of SSDs is strongly influenced by two special properties of NAND flash memory: memory cells have to be erased in large numbers simultaneously (since flash blocks are the erase unit) before writing (known as programming) and the number of *program/erase (P/E) cycles* is limited to several thousands (depending on the size of the cell, flash medium type, and the programming algorithm). As a result, flash pages are updated *out-of-place* and the limited *write endurance* of memory cells is addressed by a *wear leveling* mechanism, which strives to balance erase operations over all blocks. However, out-of-place updates demand decoupling the physical address of a page from its logical address through a (logical to physical) *address mapping policy* and also require a *garbage collection (GC)* mechanism that reclaims the space occupied by outdated pages.

Both mechanisms impact the performance: the applied address mapping policy determines the physical data layout (depending on previous writes) that affects the degree of internal parallelism of data accesses (and, hence, the overall performance) (cf. [2, 13]). The GC relocates pages with valid data before block erasure (which also occupies hardware resources of an SSD), incurring additional page reads and writes (known as *write amplification (WA)*) besides the reads and writes issued to an SSD by the host. The WA must be kept low to provide high write performance and prolong the NAND flash memory life time. One effective approach is to guarantee a substantial amount of free flash pages at any time by *over-provisioning (OP)* [7, 11], hence, sacrificing usable storage capacity. Moreover, the address mapping policy also influences the overhead of GC because it affects the internal fragmentation [1] of flash blocks.

In principle, SSDs achieve the higher performance the more page reads/writes (and also block erasures) can be performed concurrently. Due to this, SSD designers usually strive to spread page reads and writes over the available hardware resources by employing an address mapping policy similar to data striping in order to balance the utilization of hardware resources as equally as possible. However, the main difference between data striping and address mapping policies used by SSDs is that (in contrast to static mapping in data striping as used in RAIDs) a logical block can be (dynamically) mapped to any flash page within a hardware resource like a plane or a die [2]. While such dynamical address mapping policy permits to consider the utilization of hardware resources in the event of a write by scheduling a page write to a less busy hardware resource [13], this is not possible for reads (without having multiple copies of the data) since the location of the data is determined by previous writes (i.e., the workload history [11]). However, in case of writes, the number of simultaneously programmable flash pages is po-

tentially lower than the number of available planes due to power constraints [14, Sect. 7.2] (which are more restrictive for disk form factor SSDs compared to PCI Express SSDs) or due to insufficient channel bandwidth (making impossible to transfer enough data to keep all usable planes busy). Consequently, such limitations will restrict the maximum write throughput independently of the request size.

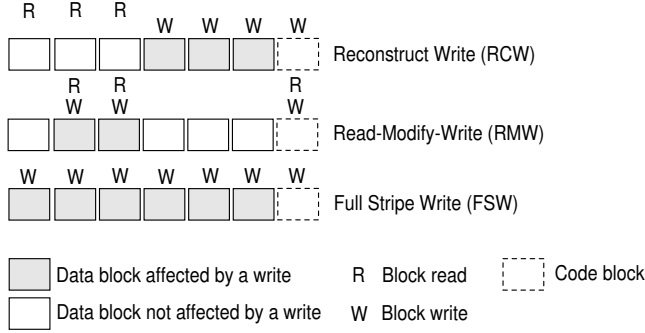
Crucial for the random access throughput of an SSD is that it is insensitive to the request size if enough concurrent whole-page read/write requests are available to keep all hardware resources busy and if the utilization of hardware resources is equally balanced as possible. In this case, an SSD can achieve the same random access throughput in face of a (sufficiently) large number of simultaneously issued single page requests as for a smaller number of requests, each affecting a correspondingly larger number of pages.

In practice, the number of concurrent page read/write requests available at a time is potentially limited by the maximum queue depth supported by the host interface (e.g., NVMe Express, SAS, SATA). This limitation occurs if the maximum queue depth is smaller than the number of pages that can be simultaneously read and/or written. In case of writes, this effect can be mitigated by using write buffering (doing so, completion is signaled to the host as soon as the data was transferred to SSD's DRAM buffer, which frees a queue slot for a new request). As a result, if an SSD employs write buffering, its throughput for random writes can surpass that of random reads (albeit reading a NAND flash page takes notably less time than programming it).

Read and write requests issued to an SSD can result in sub-page reads and writes, i.e., reading or writing less data than a flash page comprises. The performance impact of random sub-page accesses differs significantly for reads and writes. In case of random reads, this will result in a lower throughput (compared to whole-page reads), because the whole flash page has to be read from NAND flash memory despite of the fact that only a fraction of its data amount was requested. In case of random writes, the throughput of a random sub-page write depends on whether the referenced flash page already contains valid data or not. If the page contains valid data, it has to be read first (in whole). Then, the (updated) data is written to a another (free) flash page. Thus, random sub-page overwrites will yield lower throughput compared to random sub-page writes not referencing a page with valid data (which can be slightly faster than writing a whole page due to lower amount of data to transfer). Moreover, since even sub-page random writes not referencing a page with valid data occupy the capacity of a whole flash page, they reduce the effective degree of OP and the SSD life time compared to whole-page random writes.

### 3. STRIPING WITH ERASURE CODING

To counter the risk of data loss and to maintain availability in the event of drive failures, drive array organization schemes, especially RAID, employ redundancy. Primarily, two techniques are used: *erasure coding*, which adds information allowing the reconstruction of data in case of drive failures, and *replication* (also known as *data mirroring*), which stores multiple copies of the data on different drives.



**Figure 1: Illustration of partial stripe writes and a full stripe write (FSW). For partial stripe writes, two code block update methods exist: a reconstruct write (RCW) and a read-modify-write (RMW).**

For larger drive arrays, erasure coding provides a much better space efficiency than replication. Erasure coding is usually combined with *data striping* such that each stripe comprises multiple *data blocks* (i.e., *stripe units* that accommodate data) and at least one *code block* (i.e., stripe unit that contains erasure code for that stripe). However, the main drawback of data striping with erasure coding is a high overhead for small writes (known as *small write penalty*), which is especially pronounced for random writes. Due to this, we devote particular attention to the random write performance and, in particular, to the random write throughput.

### 3.1 Random Write Penalty

When data striping is combined with erasure coding, updating the content of any number of data blocks within a stripe requires re-computing all erasure codes and, thus, also updating the stripe's code blocks. Depending on the starting address and the request size of a logical write (i.e., a write to a drive array) as well as on the data striping configuration (i.e., *stripe width*, which refers to the number of data/code blocks per stripe, and the *stripe unit size*), a logical write potentially incurs also physical read requests (cf. [3]).

As shown in Fig. 1, a logical write can affect either a part of the data blocks within a stripe (denoted as *partial stripe write*) or all of them (referred to as *full stripe write* (FSW)). In case of a partial stripe write, two *code block update methods* exist: either all remaining data blocks of the respective stripe are read for erasure code computation, which is referred to as *reconstruct write* (RCW), or the old versions of the data blocks to be written and the code blocks are read, which is known as *read-modify-write* (RMW) (cf. [28]). In both cases, new versions of the respective data blocks and the code blocks are written subsequently. For independent random writes, it is very likely that neither the needed data block(s) nor the code block(s) are cached. Thus, for the sake of efficiency, the code block update method that requires less reads is preferable. In case of a full stripe write, the code blocks can simply be calculated from the new version of the data blocks to be written and, thus, no reads are required.

In the following, we assume that each stripe contains  $(n - k) \geq 2$  data blocks and  $k \geq 1$  code blocks which are spread

over  $n$  different drives (resulting in a stripe width of  $n$ ). When updating  $d \geq 1$  data blocks of the stripe, RMW requires to read the old versions of  $d + k$  data and code blocks, respectively, while RCW reads the  $n - k - d$  remaining data blocks. Thus, RMW is superior to RCW if  $d + k < n - k - d$  which implies

$$d < \frac{n - 2k}{2} \quad (1)$$

and that the disk array contains at least  $n > 2k + 2$  drives for RMW to be used at all. Besides the overhead of reading old or additional blocks, respectively, modified and new blocks need also to be written. Thus, in case of RMW, updating  $d$  data blocks causes  $2 \cdot (d + k)$  read and write requests altogether. The disproportion between the number of logical write requests issued and the number of implied physical (read and write) requests is called *small write penalty* as it is most pronounced for logical write requests that comprise only a single data block.

Usually, each stripe contains more data blocks than code blocks, but since writing any of the former requires a complete update of the latter, the update frequency of code blocks is usually significantly higher than for any individual data block and it reaches the maximum if only a single data block is written at a time. As a consequence, drives solely responsible for storing code blocks would experience a higher load in terms of read/write requests. In order to avoid these drives becoming bottlenecks, code blocks are usually spread, similar to data blocks, over available drives in a round robin manner leading to a better load distribution. For homogeneous drive arrays, this is the primary reason why RAID schemes with distributed code blocks such as RAID-5 and RAID-6 are preferred over RAID-4 that uses a dedicated drive for storing code blocks (and is not used in practice anymore). However, when considering heterogeneous storage devices with significantly different performance characteristics, this changes fundamentally. In this case, it is desirable to foster non-uniform load distributions that utilize each drive proportional to its speed in order to exploit the potential of faster storage technologies.

### 3.2 Random Write Throughput

Besides the influence of the used device type and model, the random write throughput of a drive array using data striping with erasure coding essentially depends on the striping configuration and the workload characteristics. In order to analyze their impact, we quantify the random write throughput of a drive array in relation to the random access throughput of a single drive. In particular, we are interested in the speedup achieved by the drive array with respect to its configuration and workload.

Therefore, we still assume that each stripe contains  $k$  code blocks and  $n - k$  data blocks of the same size (i.e., stripe unit size) that are spread to  $n$  different drives. Then, a logical random write (to the drive array) may affect  $1 \leq d \leq n - k$  logically adjacent data blocks of a stripe. Depending on the request size and the alignment of the starting address, a logical random write may also affect multiple stripes and cover at most two data blocks of a stripe only partially, i.e., the blocks at the begin and end, respectively. However, in the following analysis, we neglect the case that data blocks

are covered only partially and assume that a logical random write affects  $d$  complete data blocks within a stripe.

Let  $\bar{\lambda}^{\text{rr}}$  and  $\bar{\lambda}^{\text{rw}}$  be the average random read throughput and the average random write throughput of a single disk, respectively, that is measured in blocks. Thus, the disk needs  $r/\bar{\lambda}^{\text{rr}}$  to randomly read  $r$  blocks and  $w/\bar{\lambda}^{\text{rw}}$  to randomly write  $w$  blocks which results in a combined throughput of  $(r/\bar{\lambda}^{\text{rr}} + w/\bar{\lambda}^{\text{rw}})^{-1}$  complex operations consisting of  $r + w$  consecutive random reads and writes. Consequently, an array with  $n$  drives yields the  $n$ -fold throughput of

$$\bar{\lambda}_{\text{Array}}^{\text{rw}}(d, n) = n \cdot \left( \frac{r}{\bar{\lambda}^{\text{rr}}} + \frac{w}{\bar{\lambda}^{\text{rw}}} \right)^{-1}, \quad (2)$$

where we can interpret  $r$  and  $w$  as the actual overall number of code and data blocks to physically read and write, respectively, when logically writing  $d$  data blocks in a stripe.

The speedup of the drive array compared to a single drive corresponds to the ratio between the array's throughput and the drive's throughput  $\bar{\lambda}^{\text{rw}}/d$  for also writing  $d$  data blocks randomly. Hence, the speedup is

$$S(d, n) = \frac{\bar{\lambda}_{\text{Array}}^{\text{rw}}(d, n)}{\bar{\lambda}^{\text{rw}}/d} = \frac{d \cdot n}{\alpha \cdot r + w}, \quad (3)$$

where  $\alpha = \bar{\lambda}^{\text{rw}}/\bar{\lambda}^{\text{rr}}$  is the hardware-dependent ratio between the average random write and the average random read throughput specific to the storage type and model used (e.g., HDDs or SSDs based on NAND flash memory). Based on the derived formula above, we can specify the speedup for all three variants of code block updates. For each code block update method, we have to insert the respective number of blocks that are caused to be physically read and written.

**Read-Modify-Write.** An RMW of  $1 \leq d < \frac{n-2k}{2}$  data blocks has to read the old versions of the affected data blocks together with all code blocks first. This are  $r = d + k$  reads altogether. Thereafter, the new versions of these blocks have to be written which results in  $w = d + k$  writes. Hence, for RMW, we get a speedup of

$$S_{\text{RMW}}(d, n, k) = \frac{d \cdot n}{\alpha \cdot (d + k) + (d + k)}. \quad (4)$$

**Reconstruct Write.** In case of an RCW of  $\frac{n-2k}{2} \leq d < n - k$  data blocks, the other remaining data blocks of the stripe are read which makes  $r = n - k - d$  reads. Thereafter, the new data blocks together with the newly computed code blocks are written which results in  $w = d + k$  overall writes. Thus, for RCW, the speedup is

$$S_{\text{RCW}}(d, n, k) = \frac{d \cdot n}{\alpha \cdot (n - k - d) + (d + k)}. \quad (5)$$

**Full Stripe Write.** An FSW results in writing all  $d = n - k$  data blocks of a stripe together with  $k$  new code blocks. It is not necessary to read any old block, i.e.,  $r = 0$ . Hence, for FSW, the speedup is

$$S_{\text{FSW}}(d, n, k) = \frac{d \cdot n}{0 + d + k} = \frac{d \cdot n}{n} = d. \quad (6)$$

Please note that we can also derive the speedup  $S_{\text{FSW}}$  of FSW by inserting  $d = n - k$  into Eq. 5, the formula of the RCW speedup. This shows that FSW is a special case of a reconstruct write that requires no reads for reconstructing the code blocks as they can be computed completely from the data blocks to be written.

When further analyzing the speedup of different code block update methods, it is, thus, sufficient to only use Eqs. 4 and 5 for RMW and RCW, respectively, since RCW already includes FSWs for  $d = n - k$ . By comparing both equations, we see that the speedup of RMW is higher than that of RCW if

$$\frac{d \cdot n}{\alpha \cdot (d + k) + (d + k)} > \frac{d \cdot n}{\alpha \cdot (n - k - d) + (d + k)}. \quad (7)$$

We can solve the above inequality for  $d$ , which gives

$$d < \frac{n - 2k}{2}. \quad (8)$$

There are two things to notice. First, the criterion does not depend on  $\alpha$  and, thus, is independent of the storage technology used. Second, this is exactly the condition under which RMW requires to read less blocks than RCW when  $d$  data blocks of a stripe are logically written. As additional reads for computing erasure codes belong to the drive array's overhead when compared to the actual number of data blocks physically written, it is clear that reducing this overhead maximizes the speedup. Hence, for  $d \geq \frac{n-2k}{2}$ , RCW starts to outperform RMW and its speedup grows further with increasing  $d$ . It reaches the maximum for  $d = n - k$  (as  $d$  is maximal for the striping configuration) which corresponds to a FSW requiring no additional reads at all.

Based on the derived formulas above, we can now calculate the speedup for random write requests that is achieved by a drive array when its striping configuration is known, i.e., number of drives and erasure codes. But conversely, it is still unclear which striping configuration is best suited for which request characteristics (e.g., request size) and how this decision depends on the particular device type and model.

## 4. DEVICE-AWARE DATA STRIPING

In this section, we analyze the impact of different device types and their inherent characteristics on the performance of a drive array using data striping with erasure coding. It is necessary to precisely understand and quantify the array's behavior as it may differ fundamentally for various storage technologies. Based on this analysis, we give recommendations how to set up the striping configuration for optimal performance. In the following, we focus on homogeneous drive arrays consisting of either HDDs or SSDs (based on NAND flash memory) and analyze these arrays with regard to the random write throughput.

### 4.1 Hard Disk Drives

Conventional HDDs exhibit a largely symmetric read and write performance. Basically, without cache effects, data is read as fast as it is written. This is because relevant performance parameters of an HDD (i.e., command overhead, rotational speed, average media data rate, and host interface data rate) are independent of the data direction, however,

with the exception of a negligibly higher average seek time for writes compared to reads [10, Sect. 23.2.4]. Nevertheless, we can safely assume that the ratio  $\alpha$  of the average random write throughput  $\bar{\lambda}^{\text{rw}}$  and the average random read throughput  $\bar{\lambda}^{\text{rr}}$  is 1 for any given request size. This simplifies Eqs. 4 and 5 to

$$S_{\text{HDD,RMW}}(d, n, k) = \frac{d \cdot n}{d + k + d + k} = \frac{d \cdot n}{2 \cdot (d + k)}, \quad (9)$$

$$S_{\text{HDD,RCW}}(d, n, k) = \frac{d \cdot n}{n - k - d + d + k} = d. \quad (10)$$

Please note that the speedup  $S_{\text{HDD,RCW}}$  of RCW corresponds to the number of data blocks  $d$  that are written when updating the stripe. This is caused by distributing the data blocks to different drives while all other remaining drives are also utilized by a similar amount of work, i.e., by either writing a code block or by reading one of the other data blocks that are still missing to calculate the erasure codes. Furthermore, when comparing Eqs. 6 and 10, the kinship of RCW and FSW becomes apparent. The highest speedup possible is achieved for a maximum number  $d = n - k$  of data blocks written in a stripe, i.e., for an FSW.

The reference unit for the speedup calculations is the size of a single data/code block in a stripe, i.e., the stripe unit size. Thus, if a logical random write request should affect more data blocks of a stripe, the request must be larger or, conversely, the size of the data blocks has to be smaller. However, comparing speedup values for different block sizes directly with each other is problematic. It is only feasible if the request size by which the data is physically read from or written to disk has no influence on the throughput of a drive. Unfortunately, for HDDs, this is not the case.

The average time, an HDD needs for serving a random read or random write request, is basically the sum of two terms: the first term corresponds to the average time  $\bar{t}_{\text{pos}}$  required to position the read/write head above the correct disk sector. Hence, by  $\bar{t}_{\text{pos}}$ , we subsume command overhead, seek time, rotational latency etc. (cf. Sect. 2.1 for details). Thereafter, the disk can start to consecutively read or write the data. The second term corresponds to the time needed to finish the read or write operation. Obviously, this time depends on the amount  $x$  of data to read or write as well as on the average data transfer rate  $\bar{R}$  which is limited by the media data rate, host interface data rate etc. (cf. Sect. 2.1 for details). The disk's throughput is then given by the product of request size  $x$  and the reciprocal of the average service time for performing the request:

$$\bar{\lambda}_{\text{HDD}}^{\text{rr}}(x) = \bar{\lambda}_{\text{HDD}}^{\text{rw}}(x) = x \cdot \left( \bar{t}_{\text{pos}} + \frac{x}{\bar{R}} \right)^{-1}. \quad (11)$$

For small requests, the service time is dominated by the positioning time of the read/write head. When neglecting the transfer time and, thus, approximating the disk's throughput by  $x \cdot (\bar{t}_{\text{pos}})^{-1}$ , its linear dependence to the request size becomes apparent. With growing request size  $x$ , the disk's throughput grows proportionally. For larger requests, however, the transfer time cannot be neglected anymore and, from Eq. 11, we see that the throughput asymptotically approaches its maximum which is limited by the disk's transfer rate  $\bar{R}$ . The average positioning time  $\bar{t}_{\text{pos}}$  and the data

transfer rate  $\bar{R}$  are specific to a particular HDD model. Both parameters can be reliably measured in experiments with a sufficient accuracy (cf. Sect. 6.2.1 for details).

Together with the speedup formulas for RMW (Eq. 9), RCW (Eq. 10), and FSW (Eq. 6), we can now predict the performance of a drive array with  $n$  disks and  $k$  erasure codes, when a logical random write request of size  $s$  is split in  $d$  data blocks on different drives. By multiplying the speedup value for the respective code block update pattern with the throughput of a single drive for requests of size  $x = s/d$ , the predicted throughput of the drive array is given by

$$\bar{\lambda}_{\text{HDD}}^{\text{rw}}(d, n, k, s) = \begin{cases} \frac{d \cdot n}{2 \cdot (d + k)} \cdot \bar{\lambda}_{\text{HDD}}^{\text{rw}}\left(\frac{s}{d}\right) & \text{if } d < \frac{n - 2 \cdot k}{2}, \\ d \cdot \bar{\lambda}_{\text{HDD}}^{\text{rw}}\left(\frac{s}{d}\right) & \text{otherwise.} \end{cases} \quad (12)$$

This way, we can analyze the throughput gained by the drive array when using the different code block update methods and, hence, find out the optimal number  $d_{\text{opt}}$  of data blocks into which the logical random write request has to be split in order to achieve the best performance. To better understand these implications and, thus, the behavior of the drive array, we distinguish three cases for comparing random write requests to each other. Each case corresponds to a possible combination of code block update patterns.

**RMW.** Two logical random write requests of the same size  $s$  that cause  $d_1 < d_2$  and  $d_2 < \frac{n - 2 \cdot k}{2}$  physical data blocks to be written, respectively, are both carried out with RMW. The first request achieves a higher throughput than the second request if

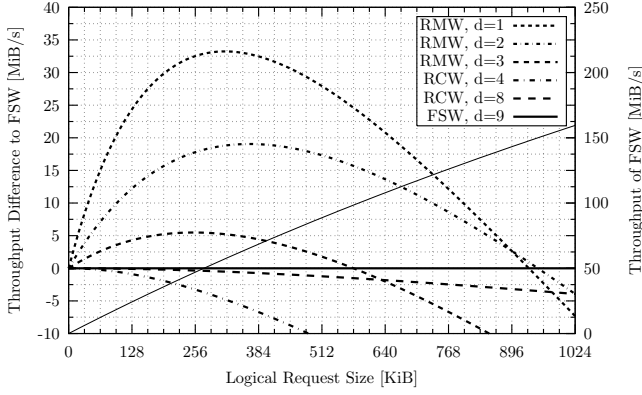
$$\frac{d_1 \cdot n}{2 \cdot (d_1 + k)} \cdot \frac{s/d_1}{\bar{t}_{\text{pos}} + \frac{s/d_1}{\bar{R}}} > \frac{d_2 \cdot n}{2 \cdot (d_2 + k)} \cdot \frac{s/d_2}{\bar{t}_{\text{pos}} + \frac{s/d_2}{\bar{R}}}. \quad (13)$$

When solving the inequality for  $s$ , we get

$$s < \frac{d_1 \cdot d_2}{k} \cdot (\bar{R} \cdot \bar{t}_{\text{pos}}), \quad (14)$$

which is interesting since the logical request size  $s$  is constrained by the product of two terms. The first term  $\frac{d_1 \cdot d_2}{k}$  is a coefficient specific to the stripe configuration and the numbers of data blocks written, while the second term  $\bar{R} \cdot \bar{t}_{\text{pos}}$  is specific to the particular HDD model. We can conclude from Ineq. 14 by inserting  $d_1 = 1$  and  $d_2 = 2$  that, in case of RMWs, it is better to not split small logical random writes at all that are smaller than  $2 \cdot k^{-1} \cdot \bar{R} \cdot \bar{t}_{\text{pos}}$ . The drive's ability to read and write larger data blocks much more efficiently compensates and even excels the lower speedup when not writing to more disk's in parallel. Only, if the logical random write requests become larger than  $d \cdot (d - 1) \cdot k^{-1} \cdot \bar{R} \cdot \bar{t}_{\text{pos}}$ , it is worthwhile to split the request into  $d \geq 2$  data blocks on different disks.

**RCW/FSW.** For two logical random writes of size  $s$  that both affect  $d_1, d_2 \geq \frac{n - 2 \cdot k}{2}$  data blocks of a stripe, respectively, RCW or FSW is used. As we can use the same formula (cf. Eq. 12) to predict the throughput of both code block update methods, we do not have to distinguish them in the following. The first logical random write affecting  $d_1$  physical data blocks has a higher throughput than the



**Figure 2: Analytical average random write throughput of RMW a/ RCW compared to FSW in an array of Seagate ST91000640SS HDDs ( $n=10$  and  $k=1$ ).**

second write with  $d_2$  blocks if

$$d_1 \cdot \frac{s/d_1}{\bar{t}_{\text{pos}} + \frac{s/d_1}{\bar{R}}} > d_2 \cdot \frac{s/d_2}{\bar{t}_{\text{pos}} + \frac{s/d_2}{\bar{R}}} \quad (15)$$

We can simplify the above inequality to

$$d_1 > d_2. \quad (16)$$

Hence, when using RCW, it always pays off to distribute physical writes to more disks. Moreover, since FSW uses the maximum number  $n - k$  of available data blocks in a stripe, it is, thus, superior to all other RCW variants for any given logical request size.

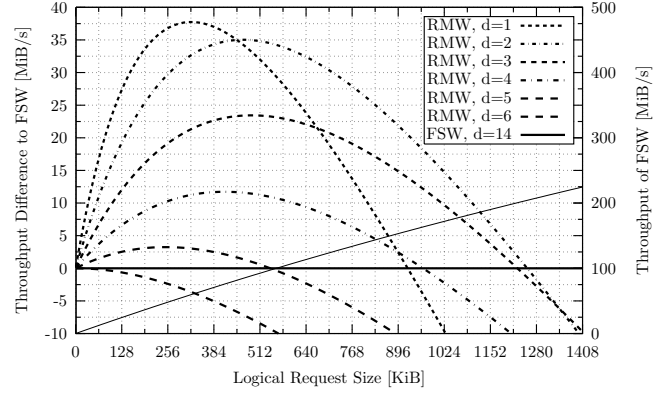
**RMW vs. RCW.** In the last case, we compare the achievable throughput for logical random write requests of size  $s$  when using RMW and RCW with  $d_1 < \frac{n-2 \cdot k}{2} \leq d_2$  data blocks, respectively. Please note that the latter (i.e., RCW) also includes FSW if  $d_2 = n - k$ . RMW has a higher throughput than RCW if

$$\frac{d_1 \cdot n}{2 \cdot (d_1 + k)} \cdot \frac{s/d_1}{\bar{t}_{\text{pos}} + \frac{s/d_1}{\bar{R}}} > d_2 \cdot \frac{s/d_2}{\bar{t}_{\text{pos}} + \frac{s/d_2}{\bar{R}}} \quad (17)$$

Similar to the previous cases, we can simplify the above inequality and solve it for  $s$ . Although, it takes a little bit more effort this time, we finally get

$$s < \frac{d_1 d_2 \cdot (n - 2d_1 - 2k)}{2d_1 d_2 + 2d_2 k - d_1 n} \cdot (\bar{R} \cdot \bar{t}_{\text{pos}}), \quad (18)$$

which states that it is better to concentrate logical random writes with a size smaller than the right side of Eq. 18 with RMW to a few disks in order to write larger physical blocks more efficiently. By inserting  $d_2 = n - k$  into Eq. 18, we can also determine up to which request size RMW is superior to FSW. The logical request size  $s$ , up to which RMW is preferable to RCW and even FSW, is again determined by a coefficient, specific to striping configuration and number of affected data blocks, and by the hardware specific product of data transfer rate  $\bar{R}$  and average head positioning time  $\bar{t}_{\text{pos}}$  of the given HDD model.



**Figure 3: Analytical average random write throughput of RMW compared to FSW in an array of Seagate ST91000640SS HDDs ( $n=16$  and  $k=2$ ).**

Leveraging the derived formulas above, we conduct an exemplary analysis of two drive arrays consisting of Seagate ST91000640SS HDDs. This drive model is also used throughout the evaluation in Sect. 6. We measured the HDD's average head positioning time and average data transfer rate to be 5 ms and nearly 89,592 KiB/s, respectively (cf. Sect. 6.2.1 for details). Based on these drive-specific values, Fig. 2 shows the predicted throughput for an array of  $n = 10$  drives with  $k = 1$  erasure codes, while Fig. 3 plots the predictions for  $n = 16$  drives with  $k = 2$  erasure codes. For the sake of clarity, we only present curves for selected values of affected data blocks  $d$  and compare the throughput gained by the implied code block update method to the performance of FSW for logical random writes of varying request sizes. Thus, the curves quantify the throughput gains and losses compared to FSW on the  $y_1$ -axis (left), while the absolute throughput of FSW is measured on the  $y_2$ -axis (right). For the graphed interval of logical request sizes, the throughput of FSW grows nearly linearly.

For the smaller drive array, as shown in Fig. 2, RCW is applied when  $4 \leq d \leq 8$  data blocks of a stripe are affected by a logical random write. Thus, the curves for  $d \in \{4, 8\}$  determine the lower and upper bound for the throughput achievable by RCW, respectively. Both are unfavorable for all logical request sizes when compared to FSW which illustrates Eq. 16. Contrarily, all RMW variants ( $d \in \{1, 2, 3\}$ ) exhibit a larger throughput than FSW for small requests and, especially, RMW with  $d = 1$  performs best over a wide range of request sizes. In particular, it is superior to FSW for logical requests up to a size of 930 KiB which we can calculate easily by using Eq. 18. Please note that there is a small interval of request sizes from 896 KiB to 949 KiB, calculated with Eqs. 14 and 18, respectively, for which RMW with  $d = 2$  performs slightly better than both FSW and RMW with  $d = 1$ . As the throughput gain is not substantial, we may be tempted to neglect this case. However, for different striping configurations and, especially, more drives, RMWs with  $d > 1$  become relevant, too. For instance, for the larger array with 16 drives and 2 erasure codes as shown in Fig. 3, RMW with  $d = 2$  is the best choice for logical request sizes between 448 KiB and 1254 KiB.

Summarizing our analysis, up to a certain logical request size  $c_{\text{fsw}}$ , the highest aggregated random write throughput is achieved in case of RMW and for requests greater than this size, FSW results in the highest throughput. Depending on the number of drives and the number of used erasure codes, the request size interval, where RMW is the best strategy, is divided in one or more subintervals meaning that in the first subinterval, RMW affecting only a single data block is best, in the second subinterval, RMW affecting two data blocks is best etc. Usually, the number of subintervals is much lower than  $\frac{n-2k}{2}$  meaning that RMWs affecting more than a rather low number of data blocks never deliver the highest throughput. In that case, either an RMW affecting less data blocks or FSW performs better.

This insight allows to choose the best performing RMW variant (e.g., RMW affecting a single data block) provided that almost all logical requests have a size for that a single of those methods provides the highest random write throughput. Assuming a fixed stripe width  $n$  and a fixed number of erasure codes  $k$ , the number of data blocks affected by an RMW can only be influenced by the choice of the stripe unit size  $u$ . This means that the stripe unit size should be set such that as many as possible logical writes will end up in RMWs writing exactly the desired number of data blocks.

In contrast to this, when (almost) all logical random writes have a certain request size that is greater than  $c_{\text{fsw}}$ , the stripe unit size should be tuned such that logical writes end up only in FSWs. However, this is only possible if the logical requests are appropriately aligned to the stripes. The reason is that the stripe unit size cannot be chosen small enough (since it is at least bounded below by the sector size) to avoid that a logical write covers up to two stripes only partially.

For mixed random write workloads, the most favorable frequency distribution of the three code block update methods depends on the particular request size distribution. In this case, the choice of the stripe unit size resulting in the highest aggregated random write throughput is not straightforward.

## 4.2 NAND Flash Memory SSDs

The random access throughput of an SSD is largely insensitive to the request size (cf. Sect. 2.2) if enough page requests are available all the time to keep all hardware resources busy. However, depending on the used SSD model, the maximum queue depth of the host interface can be too low to keep all hardware resources busy in face of single-page requests. In this case, the highest possible random read/write throughput is potentially only achieved for requests comprising multiple flash pages. In the following, we assume that the queue depth is sufficiently high. This is, however, not a severe restriction since the AHCI-based SATA interface still widely used today is assumed to be replaced by the NVMe interface soon. In contrast to AHCI, which offers a queue depth of only 32 (which is a limit imposed by SATA [25]), NVMe Express exhibits a queue depth of up to 65,536 [18]. Another possibility is to use SSDs with SAS interface, because SAS offers a queue depth of up to 254 [27].

The insensitivity of the random access throughput of an SSD to the request size, however, only exists for requests whose size is an integral multiple of the flash page size. Especially

random write requests that are smaller than a flash page, will result in a throughput reduction that is proportional to the ratio between the request size and the flash page size  $p$ . It is, thus, important to minimize the number of sub-page random write requests. Moreover, the requests issued to an SSD should be aligned to page boundaries, because a request with a size of a page that is not page-aligned leads to two sub-page requests resulting in two SSD internal RMW operations. Larger non-aligned requests with a size of  $a$  sectors per page, lead to  $\lfloor \frac{a \cdot 512 \text{ bytes}}{p} \rfloor$  complete flash pages written and two sub-page writes (to the first and the last page affected by the write). This means that the larger the request is, the smaller the penalty of non-aligned write requests becomes.

For SSD arrays using data striping with erasure coding, the insensitivity of random access throughput to the physical request size can be exploited to increase the aggregated random write throughput. Assuming a (nearly) constant random write throughput per SSD, the resulting aggregated random write throughput of an SSD array is directly proportional to the speedup of the used code block update method. We already showed in Sect. 3.2 that the speedup of FSW is the highest, followed by RCW and, then, RMW. Therefore, random writes comprising multiple flash pages should be distributed over several SSDs. Ideally, a random write should cover all  $(n - k)$  data blocks of a stripe resulting in an FSW (and making reads unnecessary) rather than in an RCW or even an RMW leading to a lower throughput because of the required reads. This suggests to set the stripe size and, thus, the stripe unit size as small as possible to increase the chance of getting more FSWs.

However, because the physical writes to the individual SSDs should not become smaller than a flash page to avoid sub-page writes, the stripe unit size should not be set to a value smaller than the flash page size. This also avoids that logical random writes with a request size below the flash page size would be split in even smaller parts written to multiple stripe units heavily affecting performance and SSD endurance. Due to a similar reason, the stripe unit size should be set to an integral multiple of the flash page size implying a stripe size that is an integral multiple of  $(n - k) \cdot p$ . In that case, logical writes whose size is an integral multiple of the stripe size and that are stripe-aligned will result solely in FSWs, while those that are not stripe-aligned or whose size is only an integral multiple of the flash page size, will result in a number of FSWs and in two partial stripe writes (for the first stripe and the last stripe affected by the write, respectively) that are either an RMW or an RCW. Moreover, each of the two partial writes causes one sub-page write if the request is not aligned to page boundaries. Again, the larger the logical request is, the smaller the effect of the disadvantageous partial writes and the sub-page writes become.

Summarizing, we suggest to set the stripe unit size equal to the flash page size if the vast majority of the requests are page-aligned. The rationale behind this is that this increases the probability that multi-page random writes will affect a larger number of data blocks (compared to larger stripe units) and will more likely end up in an RCW or FSW instead of an RMW (which also depends on the stripe width  $n$ ). The chance of getting page-aligned requests can

be increased by aligning device partitions and file systems accordingly [11]. Even better is if requests are aligned to stripe boundaries and if their size is an integral multiple of the stripe size. If a larger part of the requests is not page-aligned, even smaller requests could hit two stripe units requiring two partial stripe writes involving sub-page writes. In that case, setting the stripe unit size to a small multiple (e.g., 2-, 4-, or 8-fold) of the flash page size can be beneficial to restrict small and medium-sized requests to hit a single stripe unit. Choosing the data striping configuration requires in any case knowing the (effective) flash page size of the used SSD model. The flash page size can easily be determined by measuring the latency of random reads [2].

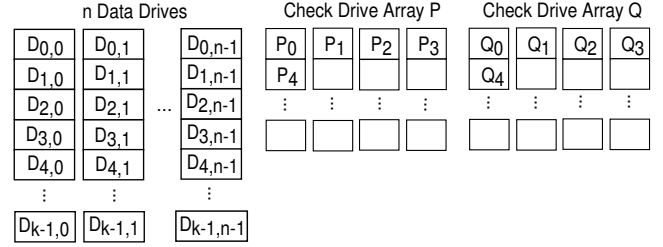
## 5. CODE BLOCK REQUEST OFFLOADING

In order to leverage the potential of faster storage technologies in heterogeneous drive arrays, it is sensible to utilize each drive proportional to its speed. This prevents bottlenecks and maximizes the throughput. Non-uniform load distributions that spread the requests to the drives in proportion to their relative performance can be attained by placing those blocks that are accessed more frequently and/or an appropriately larger fraction of blocks onto faster devices. As code blocks are updated more frequently than data blocks (cf. random write penalty in Sect. 3.1), however, it makes only sense to additionally place data blocks onto faster devices if all code blocks already reside there. Shifting the code blocks from slower devices to faster ones offloads the corresponding code block requests, thus, allowing the slower devices to perform more data block requests in the same time period, which potentially increases performance. We denote this approach as *Code Block Request Offloading (CBRO)*.

We now describe a specific drive array organization scheme that applies CBRO by placing all data blocks on slower devices and all code blocks on faster devices. Please note that the drive array organization described is not as effective in case of RCW and FSW as for RMW, because the update frequency of code blocks decreases in case of RCW and the code blocks have not to be read at all for FSW. However, based on our device-aware data striping analysis in the last section, we strongly recommend tuning the data striping configuration to foster RMWs for particular workloads and device types, e.g., small random writes on HDD arrays. This makes CBRO an ideal addition to those configurations.

### 5.1 Drive Array Organization

With CBRO, a drive array contains homogeneous *data drives* that accommodate data blocks, and one *check drive array (CDA)* per erasure code that accommodates all related code blocks (spreading code blocks belonging to different erasure codes over CDAs provides no performance benefit since all code blocks related to a data block must be updated when writing). Please note that a single device can also fulfill the role of a CDA, whenever it can provide enough capacity to store the code blocks and enough performance to serve all requests issued to these. Besides, CBRO can also be recursively applied to CDAs. An exemplary drive array using CBRO and two erasure codes  $P$  and  $Q$  is shown in Fig. 4, whereby the code blocks related to the erasure codes are accommodated by two striped CDAs each comprising 4 drives.



**Figure 4: Example drive array using two erasure codes and CBRO. Stripe  $i$  comprises  $n$  data blocks  $D_{i,j}$  as well as two code blocks  $P_i$  and  $Q_i$  each stored in separate (striped) check drive arrays.**

### 5.2 Random Write Performance

As already mentioned, CBRO pays off when used in conjunction with RMW requests. The performance analysis, thus, focuses on this code block update variant. With RMW, a logical random write affecting  $d$  (adjacent) complete data blocks within a stripe causes  $d + k$  physical read and subsequent write requests when  $k$  erasure codes are used. With CBRO,  $k$  read and write requests are shifted to  $k$  different CDAs leaving  $d$  read and write requests to be performed by  $d$  out of  $n$  data drives (resulting in an effective stripe width of  $n + k$ ). If all CDAs are able to cope with code block updates from all data drives, the aggregated random write throughput of CBRO is independent of  $k$ . Based on our speedup formulas in Sect. 3.2, we can determine the resulting speedup of CBRO for RMW. With CBRO, only  $r = d$  reads and  $w = d$  writes have to be carried out by the data drives since all requests for code blocks have been offloaded. Thus, Eq. 3 gives a speedup of

$$S_{\text{CBRO,RMW}}(n) = \frac{d \cdot n}{\alpha d + d} = \frac{n}{\alpha + 1} \quad (19)$$

when compared to a single data drive. Please note that this number is also independent of  $d$ . Now, we can calculate the throughput speedup due to CBRO compared to a homogeneous drive array by considering the speedup ratio between  $S_{\text{CBRO,RMW}}$  and  $S_{\text{RMW}}$  from Eq. 4 which gives

$$\frac{S_{\text{CBRO,RMW}}(n)}{S_{\text{RMW}}(d, n, k)} = \left( \frac{n}{\alpha + 1} \right) \left( \frac{d \cdot n}{(\alpha + 1)(d + k)} \right)^{-1} = \frac{d + k}{d}. \quad (20)$$

Obviously, the achieved throughput speedup due to CBRO increases with the number of used erasure codes  $k$  (which, however, also requires more CDAs). For a fixed  $k$ , the highest throughput speedup due to CBRO is achieved if each logical random write affects only one data block within a stripe (i.e.,  $d = 1$ ). As an example, in the case of  $d = 1$ , the aggregated random write throughput of a drive array using CBRO is 2 times higher than of a RAID-5 setup and 3 times higher compared to a RAID-6 setup. However, for a fixed  $k$ , the throughput speedup due to CBRO will decrease with growing number of data blocks  $d$  affected by a logical random write. Please note that CBRO has no significant effect on the random write latency since this value is still bounded below by the time that a data drive requires for a single random RMW request [12].

### 5.3 Requirements on Check Drive Arrays

**Capacity.** Each CDA  $A_i$  for erasure code  $i$  has to provide a total capacity large enough to store all respective code blocks. Thus, its total capacity  $C_{A_i}$  needs to be at least as large as the used capacity  $C_D$  of a single data drive:

$$\forall i : C_{A_i} \geq C_D \quad (21)$$

**Reliability.** A CDA should be at least as reliable as a data drive to avoid reducing the overall reliability. Hence, the mean time to failure  $\text{MTTF}_{A_i}$  of CDA  $A_i$  needs to be at least as high as the  $\text{MTTF}_D$  of a data drive:

$$\forall i : \text{MTTF}_{A_i} \geq \text{MTTF}_D \quad (22)$$

**Performance.** The performance of each CDA has to be high enough to avoid becoming a bottleneck. Thus, it has to provide at least the aggregated RMW throughput of all data drives together. Let  $\bar{t}_{\text{RMW},A_i}(u)$  and  $\bar{t}_{\text{RMW},D}(u)$  be the mean service times of a single RMW with the request size  $u$  on the CDA  $A_i$  and one of the  $n$  (homogeneous) data drives, respectively, then it must hold:

$$\forall i : \frac{1}{\bar{t}_{\text{RMW},A_i}(u)} \geq \frac{n}{\bar{t}_{\text{RMW},D}(u)} \quad (23)$$

**Dependencies.** Please note that the requirements above are not independent. Together, they determine the number of drives required for each CDA. It is particularly desirable to keep this number low as faster drives used in a CDA are usually more expensive. But since faster drives often provide lower capacities, their number cannot be reduced arbitrarily. Moreover, ensuring a certain performance level and degree of reliability in a CDA may require additional drives which increase the costs again. Hence, not every feasible combination of (heterogeneous) drives (for CBRO) is economically sensible. But both, feasibility as well as costs, are primarily determined by the chosen storage technologies.

### 5.4 Technological Considerations

In the following, we examine the suitability of different storage device type combinations for CBRO with respect to their capacity and random access performance characteristics as well as their reliability.

#### 5.4.1 Capacity and Random Access Performance

HDDs exhibit a notably better (i.e., lower) total storage device cost per capacity ratio than SSDs (based on NAND flash memory). In addition to this, the RMW code block update method results in case of HDDs in the highest aggregated random write throughput for request sizes in the range of several hundreds of KiBs (depending on the used HDD model, number of erasure codes and deployed HDDs). Both, the lower costs and the superiority of RMW for larger requests, makes HDDs an ideal candidate to serve as data drives in a drive array using CBRO. However, the random access throughput difference between the slowest and fastest

Table 1: Typical page sizes of NAND flash media.

Flash Medium	Flash Page Size
SLC [8, 15]	2 or 4 KiB
MLC [9, 16]	4 or 8 KiB
TLC [21, 29]	8 KiB

available model amounts less than a half of an order of magnitude. The following example illustrates the differences between the average data transfer rates provided by fast and slow HDD models: one of the fastest enterprise-class HDDs, the HGST HUS156060VLS600, delivers a data transfer rate of up to 198 MB/s [5], while even desktop HDDs (which cost notably less and target a completely different usage scenario) can provide data transfer rates of up to 100 MB/s. Moreover, also optimizations like short-stroking cannot provide a substantial improvement of the random access throughput [6]. As a consequence, even using the slowest HDDs as data drives and the fastest HDDs in CDAs is not appealing due to the demand for a high number of HDDs in each CDA (which potentially requires using erasure coding).

SSDs based on NAND flash memory are capable of providing a considerably higher RMW throughput than HDDs, especially for small requests [12], which makes them suitable for CDAs in combination with HDDs as data drives. However, the random access performance of SSDs depends strongly on the used flash medium type as well as on their system-level and flash-level architecture (cf. Sect. 2.2). In case of larger requests, the used host interface as well as the form factor of an SSD have strong impact on the achievable RMW throughput. Especially disk form factor SSDs with a disk host interface like SATA or SAS, potentially provide less than a half of an order of magnitude higher RMW throughput for larger requests than HDDs. As a consequence, a larger number of SSDs has to be deployed per CDA in order to meet the requirements on performance, which is potentially exacerbated when additional SSDs are necessary to also meet reliability demands.

Using SSDs as data drives is only beneficial for small requests, i.e., with a request size corresponding to the size of a few flash pages and depending on the number of drives, because for larger requests RMW will yield lower aggregated random write throughput than RCW and FSW (cf. Sect. 4.2). Thus, the usage of SSDs as data drives is only applicable for a smaller range of applications. However, the page size of SSDs (and usually also the per device capacity) depends significantly on the used flash medium type. SSDs based on TLC or QLC flash have often larger flash pages than SSDs based on SLC or MLC (see Table 1) as well as larger capacities. As a result, it can be beneficial to use TLC/QLC SSDs as data drives in combination with SLC/MLC SSDs deployed in CDAs. Please note that TLC/QLC SSDs also often have less dies and planes than a SLC/MLC SSD with a comparable per device capacity, which results also in a potentially lower RMW throughput. However, when SSDs are used as data drives (in scenarios with small requests), it is not appealing to use HDDs in CDAs due to the fact that HDDs provide a very low RMW throughput for small requests. However, when using CBRO the most reliable

method to assess the suitability of a particular storage device combination is to conduct RMW throughput measurements (see Sect. 6.4.1). The reason is that parameter values specific to a particular storage device model (e.g., flash page size of an SSD or the average data transfer rate of an HDD) are often not included in the product documentation.

#### 5.4.2 Write Endurance and Reliability

For NAND flash SSDs limited write endurance is an issue, irrespective of whether they are used as data drives or in CDAs. The number of possible writes, more precisely program/erase (P/E) cycles, is determined by the used NAND flash package (more bits per cell and smaller feature size mostly yield less P/E cycles), and the write amplification (WA) [7], which strongly depends on the degree of OP, but also on the write request size. Previous studies [7, 11] on the influence of OP to (small) random write performance of SSDs indicate that a considerable OP (at least about 25%, which is quite common for enterprise SSDs) helps to keep the WA low and, thus, to increase the write endurance notably. However, SSDs will eventually have to be replaced (which can be done without interrupting the operation of a CBRO-based drive array). The appropriate time for replacement can be determined based on SMART [26], which is a mechanism for self-monitoring and error reporting incorporated in SSDs using standard disk drive interfaces (for PCI Express SSDs similar mechanisms are available). For most such SSDs a wear-out indicator (value of corresponding SMART attribute) can be permanently monitored in order to detect if the wear threshold defined by the manufacturer has been reached.

In the simplest case, a CDA consists of a single device, thus, its reliability is comparable with that of a data drive. Since SSDs are considered more reliable than HDDs [14, Sect. 8.7], with a single SSD CDA no precautions are needed when HDDs are used as data drives. However, in CDAs comprising multiple drives, failures become more likely with more drives, thus, potentially demanding erasure coding or data replication, which in turn increases the number of drives in the CDA. For small CDAs, no redundancy (pure data striping for load balancing), data mirroring (like with RAID-1) or a combination of both is potentially suitable in order to obtain a sufficient reliability. A recent reliability analysis of SSD arrays using a single erasure code [17] suggests that for workloads dominated by small writes and very small SSD arrays (up to 4 drives), pure data striping can be more reliable than RAID-5 due to additional writes necessary to maintain parity (especially with low OP). For such small requests that RMW is superior to RCW and FSW, a recursive application of CBRO can provide the necessary reliability with less additional devices compared to replication and data layouts with distributed code blocks like RAID-5 and RAID-6. This changes for larger requests because RCWs and FSWs result in notably higher random write throughput than RMWs (and CBRO yields almost no benefits in case of RCWs and FSWs). However, it should be noted that even drive failures within a CDA do not entail immediate data loss, but rather a (partial) loss of redundancy, and potentially lower CDA performance during a rebuild.

## 6. EXPERIMENTAL EVALUATION

This section provides an experimental evaluation of device-aware data striping and of the benefits due to CBRO. First, we describe our experimental setup. Then, we examine the sensitivity of the random access throughput to the request size for HDDs and SSDs. Afterwards, we experimentally validate the benefits of device-aware data striping in homogeneous drive arrays comprising either HDDs or SSDs. Finally, we describe an implementation of CBRO, using HDDs as data drives and SSD arrays as CDAs. Based on this, we evaluate the random write throughput increase due to CBRO and its scalability for small as well as larger requests.

### 6.1 Experimental Setup

**Hardware and OS.** All experiments were conducted on dual-socket server machines (each with two Intel Xeon E5-2680 CPU and 256 GiB RAM) equipped with Seagate HDDs and Samsung SSDs. The used Seagate ST91000640SS 1 TB HDDs (SAS-2, 7200 rpm) are attached to HBAs based on LSI SAS2308. The used Samsung 830 SSDs (SATA-3, with firmware revision CXM03B1Q) are attached to HBAs based on LSI SAS3008 and have capacities of 128 GB (referred to as *L* for low capacity) and 256 GB (referred to as *H* for high capacity). Furthermore, the HDDs were configured to use *Tagged Command Queuing (TCQ)* and SSDs to use *Native Command Queuing (NCQ)*. Gentoo Linux with kernel version 3.16.7 was used as OS, and the kernel was configured to use *noop* I/O scheduler for HDDs (leaving seek time optimization to TCQ) and SSDs.

**Software RAID and CBRO Implementation.** For homogeneous HDD and SSD RAID-5/6 setups, where the code blocks are spread over all drives (which is not the case for CBRO), the default data layout (i.e., left-asymmetric) was used. In case of CBRO we used the “parity-last” data layout, where the code blocks are placed onto the last drive(s). All experiments for homogeneous and heterogeneous RAID-5/6 setups were conducted using a customized version of the *MD RAID 4/5/6* driver (known as Linux Software RAID), which was changed such that it performs RMWs (if appropriate) also when two erasure codes are used, because the original driver performs always RCWs when two erasure codes are used. The *MD RAID 4/5/6* driver uses a stripe buffer (to process multiple outstanding write requests), whose size was chosen by setting the configuration parameter “stripe\_cache\_size” such that up to  $q$  stripes can be accommodated if possible (because the maximum value is 32,768), where  $q$  denotes the queue depth for asynchronous I/O. In addition to this, logical write request processing can be distributed to multiple threads through the configuration parameter “group\_thread\_cnt” to increase the throughput. Unless otherwise stated, we use this feature only for SSD-based RAID-5/6 setups including 1-CBRO.0-based CDAs, whereby we set the number of threads to one thread per deployed SSD. Furthermore, for all RAID-5/6 setups the write-intent bitmap was disabled, which would otherwise reduce the (random) write throughput. Please note that *MD RAID 4/5/6* demands that the striping unit size is specified in KiB and has to be a power of 2 greater or equal to

4 KiB, which imposes limitations on the possible data striping configurations.

In the remainder, we use the following naming scheme for CBRO setups, where  $k$  is the number of CDAs (equal to number of used erasure codes),  $DD$  denotes a data drive within a CDA, and  $CD$  a data drive in the nested CDA:

$$\langle k \rangle - \text{CBRO} . \langle \text{Type of 1st CDA} \rangle - \langle \#DD \rangle - \langle \#CD \rangle / \\ \langle \text{Type of 2nd CDA} \rangle - \langle \#DD \rangle - \langle \#CD \rangle$$

We deploy only SSDs in CDAs and annotate the number of drives by an  $L$  or a  $H$  to distinguish between the two versions of the Samsung 830 SSDs with 128 GB and 256 GB. In CDA setups based on an SSD RAID-0 or an SSD RAID-5, stripe unit size was set to 8 KiB, i.e., effective flash page size [12]. However, in nested CBRO setups which were used for small random writes affecting whole data blocks, the stripe unit size was set to the corresponding request size.

**SSD Preconditioning and Over-Provisioning.** In order to eliminate the influence of previous workloads, at the start of an experiment, an *enhanced ATA secure erase command* that erases the data in all flash cells was performed on all used SSDs. Afterwards, the SSDs were preconditioned in order to obtain steady state performance [11]) by writing 3 times the total physical flash capacity at random with the a request size of 8 KiB, which corresponds to the (effective) flash page size. In the case of CBRO, the degree of OP results from the number of SSDs deployed in a particular CDA setup: We used 4 SSD-H (or 8 SSD-L), 5 SSD-H (or 10 SSD-L) and 7 SSD-H (or 14 SSD-L) per CDA (as data drives), yielding 9%, 27%, and 48% OP, respectively. For experiments not regarding CBRO, the used degree of OP was explicitly specified.

**Benchmarks and Measurement Methodology.** Each measurement was repeated three times and the arithmetic average is reported as result. All measurements were performed on raw block devices under the use of native Linux asynchronous I/O and using direct I/O to minimize the impact of caching. Please note that direct I/O requires that the starting address is aligned to sector boundaries and that the request size is a multiple of the sector size. All random read/write throughput measurements were conducted with *fiio* 2.1.8 and a per device queue depth (QD) of 32 requests (since higher queue depths yield no throughput increase). For measurements of RMW throughput in the context of CBRO a custom benchmark tool was used and the results were obtained based on a duration of 10 minutes per repetition, while using a per device queue depth of 32. In addition to our previous work [12] on CBRO, we consider also larger request sizes due to the fact that in HDD arrays (using data striping with erasure coding) RMWs can be favorable for random write sizes in the range of several hundreds of KiB, depending on the used HDD model and the data striping configuration. Consequently, in the context of CBRO we consider random writes with a small request size of 4 KiB, 8 KiB, and 16 KiB as well as with a larger request size of 64 KiB, 256 KiB, and 1024 KiB.

## 6.2 Single Drive Random Access Throughput

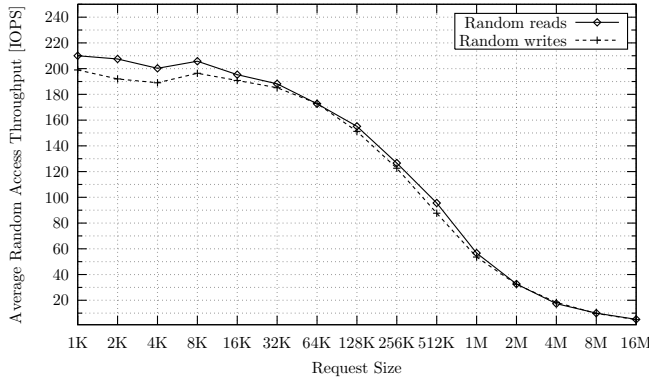
In the following, we examine the sensitivity of the random access throughput of a single HDD and SSD to the request size. For SSDs, we also consider the dependency of random access throughput to the degree of OP.

### 6.2.1 Hard Disk Drives (HDDs)

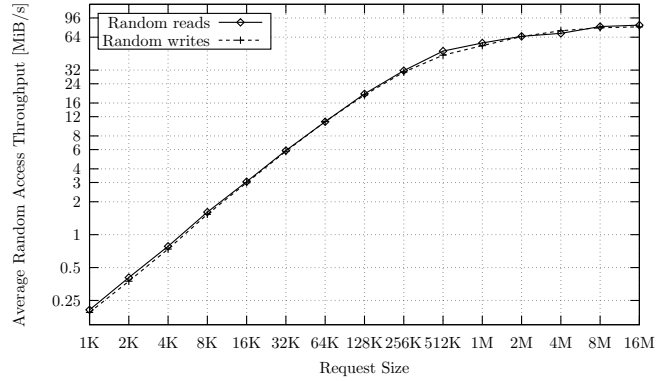
For small requests, the random access throughput of HDDs is primarily determined by the head positioning time, limiting the achievable number of IOPS. With increasing request size, the number of IOPS declines due to the increasing data transfer time (cf. Sect. 2.1). However, the random access throughput of HDDs is comparably high for reads and writes. To validate this, we conducted measurements of the average random read and write throughput (in terms of IOPS and MiB/s) for the Seagate ST91000640SS HDD for request sizes between 1 KiB and 16 MiB. The results are shown in Fig. 5, where throughput in IOPS is depicted in Fig. 5(a) and the throughput in MiB/s in Fig. 5(b).

Looking at the throughput in IOPS shown in Fig. 5(a) reveals that random reads are about 5% faster for requests up to 32 KiB. For example, in case of the smallest request size (i.e., 1 KiB), the read throughput amounts 210 IOPS while the write throughput is only 199 IOPS. This stems from a higher head settling time for writes in comparison to reads [10, Sect. 23.2.4]. However, the difference disappears for larger requests making random reads and writes comparably fast. Considering the random access throughput of HDDs in terms of MiB/s shown in Fig. 5(b), it can be noticed that up to about 128 KiB it increases nearly linearly with the request size, while the throughput curve slowly levels out for larger requests, eventually approaching the media data rate. Consequently, for small requests, the random access throughput of an HDD is primarily limited due to head positioning time, while for large requests, it is limited by the media data rate. However, albeit the small difference between read and write throughput for small request sizes, reads and writes can be considered as comparably fast.

We have to obtain the average head positioning time and the average data transfer rate of a Seagate ST91000640SS HDD through measurements to determine onto which number of HDDs should logical random writes with a certain request size be spread in order to achieve the highest aggregated random write throughput in an array (cf. Sect. 4.1). For very small requests (one sector is the smallest possible size) the data transfer time is negligible small, thus, the average service time for a request gives a reasonable approximation of the average head positioning time. We consider the reciprocal of the average write throughput in IOPS as the average service time for a request. As a result, we measured the average throughput based on 10,000,000 single-sector random writes using the whole logical address range. The measured average throughput of single-sector random writes is 200 IOPS, which corresponds to an average service time of 5 ms. We determine the average data transfer rate of an HDD by measuring the write throughput for requests with a smallest possible head positioning time, thus, for large sequential writes. Using the whole logical address range of the HDD, we got an average sequential write throughput of 89,592 KiB/s for requests comprising 16 MiB.



(a) Throughput in IOPS



(b) Throughput in MiB/s (base-2 log scale)

Figure 5: Average random access throughput of an HDD on varying request size.

### 6.2.2 NAND Flash Memory SSDs

In case of SSDs, we examine the sensitivity of the random access throughput to the request size for the used SSD models, i.e., the Samsung 830 128 GB (SSD-L) and 256 GB SSDs (SSD-H). We compare the sensitivity of the random access throughput to the request size for aged SSDs (that have been subject to random writes) and pristine SSDs. Aging is achieved by preconditioning with page-sized random writes (see Sect. 6.1) and for pristine SSDs the data in all flash cells was erased. In order to examine the sensitivity of the random access throughput to the request size, we measured the average random read and write throughput for request sizes between 1 KiB and 16,384 KiB (16 MiB). Please note that in case of writes the amount of data written per repetition (the measured values are averaged over 3 repetitions) corresponds to 1/4 of the physical flash capacity, hence, in case of pristine SSD no GC is necessary. The results are shown in Fig. 6.

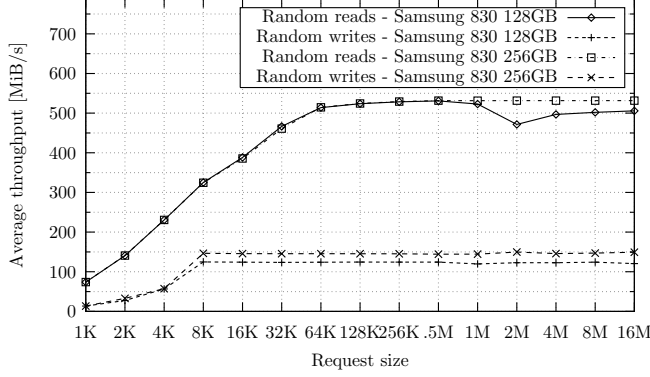
Looking at the measured throughput values for preconditioned SSDs in Fig. 6(a) reveals that the read throughput is generally higher than the write throughput, although the random access throughput for requests smaller than 8 KiB is notably lower than for larger request sizes. This is attributed to an effective flash page size of 8 KiB [12] and, thus, a lower performance due to sub-page reads and writes (cf. Sect. 2.2). Considering the write throughput for requests larger than 8 KiB, the throughput is virtually insensitive to the request size for both SSD models, while the 256 GB SSD provides a slightly higher write throughput than the 128 GB SSD. However, for whole-page reads between 8 KiB and 64 KiB the throughput depends on the request size for both SSD models, which is caused by insufficient maximum request queue depth of the SATA-3 host interface. Besides this, in case of the SSD-L, the read throughput breaks slightly down at 2 MiB, which we cannot explain.

Next, we compare the results for pristine SSDs presented in Fig. 6(b) to that of preconditioned SSDs shown in Fig. 6(a). Obviously, like for preconditioned SSDs, the read and write throughput is lower for requests smaller than 8 KiB in comparison to larger requests. Moreover, there is also a slight difference in write throughput for sub-page requests (less than 8 KiB) between the two SSD models. However, the

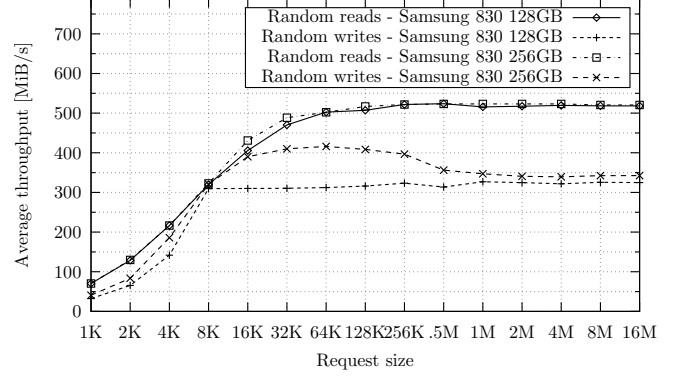
maximum random write throughput of both SSDs is notably higher than for preconditioned SSDs which can be attributed to the fact that GC is not necessary. For whole-page writes between 8 KiB and 32 KiB, the random write throughput of the 256 GB SSD increases, indicating insufficient maximum request queue depth which does not affect the 128 GB SSD in this case. Surprisingly, the random write throughput of the 256 GB SSD drops then for requests larger than 256 KiB from around 400 MiB/s to 350 MiB/s. Considering the random read throughput, both SSD models exhibit a very similar read throughput to the preconditioned SSDs, which confirms that the read throughput is independent of the degree of OP and the previous writes [11]. However, the results for preconditioned SSDs are more meaningful because they represent the condition of SSDs deployed in productive data storage systems, while pristine SSDs reflect only a temporary situation.

Even if we consider preconditioned SSDs, which represent the condition of SSDs deployed in productive data storage systems, the sensitivity of the random read throughput due to insufficient maximum request queue depth, introduces a variation of  $\alpha$  for requests between 8 KiB and 64 KiB for both Samsung 830 SSD variants. In order to illustrate this, we have derived  $\alpha$  from measurements of random write and read throughput for preconditioned SSDs. The results are depicted in Fig. 7. Looking at the course of  $\alpha$  for request sizes between 8 KiB and 64 KiB reveals that the value of  $\alpha$  declines for both SSDs. This stems from the fact that the random read throughput of each SSD model increases between 8 KiB and 64 KiB while the random write throughput remains nearly constant (cf. Fig. 6(a)). However, the variation of  $\alpha$  stems from the used disk host interface which is inadequate even for the Samsung 830 SSDs introduced in 2011. Such issues do not concern PCI Express SSDs that employ a more suitable logical device interface like NVMe Express with a maximum queue depth of 65,536 and the possibility to use even multiple queues.

However, although the random write throughput of preconditioned SSDs is independent of the request size for whole-page writes, the degree of OP determines its amount. In order to examine how the amount of random write throughput changes with the degree of OP, we also conducted write

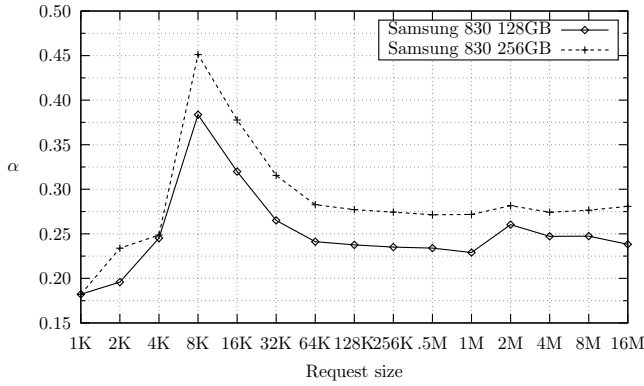


(a) Preconditioned SSDs with 27% OP



(b) Pristine SSDs

**Figure 6: Average random access throughput of preconditioned SSDs with 27% OP and pristine SSDs on varying request size (base-2 logarithmic scaled x-axis).**



**Figure 7: Ratio between the throughput of random writes and reads of Samsung 830 128/256 GB SSDs.**

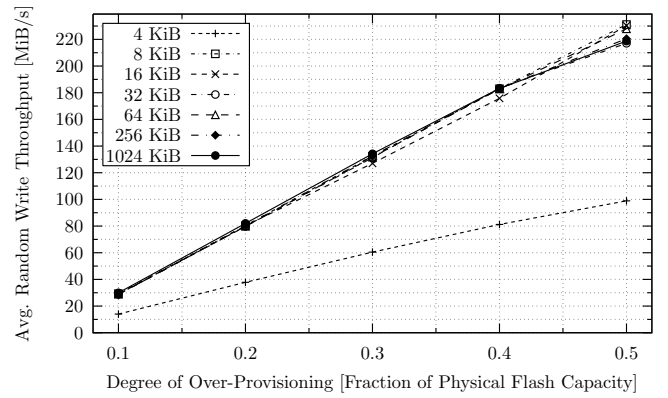
throughput measurements for preconditioned SSD-L with varying degree of OP and request size. The results are shown in Fig. 8 and indicate that the random write throughput is also virtually insensitive to the request size (except of sub-page writes represented by 4 KiB writes) when the WA changes due to a different degree of OP. Besides this, the results show that the random write throughput is strongly affected by the degree of OP. As an example, the random write throughput increases from around 30 MiB/s to 80 MiB/s, when OP is raised from 10% to 20%.

### 6.3 Device-Aware Data Striping

In this part of our experimental evaluation, we examine the benefits of device-aware data striping in homogeneous drive arrays comprising either HDDs or SSDs. In particular, we validate our analysis of the aggregated random write throughput presented in Sect. 4 and examine the performance gain due to device-aware data striping configurations in HDD as well as SSD arrays.

#### 6.3.1 Random Write Throughput of HDD Arrays

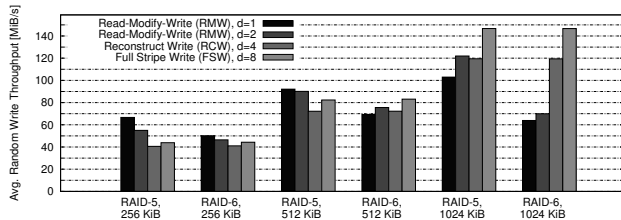
The following experimental evaluation of the random write throughput in erasure-coded HDD arrays has the objective



**Figure 8: Average random write throughput of a preconditioned Samsung 830 128 GB SSD on varying request size and degree of OP.**

to validate our analysis as well as the proposed method for the choice of the stripe unit size in Sect. 4.1.

First, we validate our analysis from Sect. 4.1, which indicates that for logical random writes up to a certain logical request size, the highest aggregated random write throughput is achieved (for arrays with  $n > 2 \cdot k + 2$  HDDs) in case of RMW. To accomplish this, we compare the average random write throughput of all three code block update methods on varying logical request sizes and when data blocks are written at whole. Considering a particular logical request size  $s$ , the used code block update method results from the number of affected data blocks  $d$  in a stripe, which depends on the stripe unit size  $u$ , stripe width  $n$ , and the number of erasure codes  $k$ . Since we consider a particular logical request size  $s$  for all code block update methods (Sect. 4.1), we have to choose the stripe width  $n$  as well as the stripe unit size  $u$  such that each code block update method can occur. Due to the constraints on the choice of the stripe unit size imposed by Linux Software RAID (Sect. 6.1), this is achieved when the number of data blocks  $n - k$  in a stripe is also a power of 2. RMW can only be favorable over RCW if  $n > 2 \cdot k + 2$  (i.e.,  $n > 4$  for  $k = 1$  and  $n > 6$  for  $k = 2$ ), hence, we



**Figure 9: Average random write throughput of HDD RAID-5 and RAID-6 on varying code block update method and request size of logical writes.**

choose  $n - k = 8$  (i.e., the smallest power of two greater than 4 and 6). This choice implies  $n = 9$  in case of  $k = 1$  (RAID-5) and  $n = 10$  in case of  $k = 2$  (RAID-6). Using the measured values of the average head positioning time and the average data transfer rate (Sect. 6.2.1) permits to calculate, for an HDD RAID-5 with  $n = 9$  and an HDD RAID-6 with  $n = 10$ , up to which logical request size RMW yields a higher throughput than FSW (based on Eq. 18) as well as to determine the logical request size intervals where which RMW variant results in the highest throughput among the others (using Eq. 14).

In case of an HDD RAID-5 with  $n = 9$  RMW is favorable for  $d$  up to 3 (since  $\lceil \frac{9-2 \cdot 1}{2} \rceil - 1 = 3$ ), however only RMW with  $d = 1$  is superior to FSW. In case of an HDD RAID-6 with  $n = 10$  RMW is favorable for  $d$  up to 2 (because  $\lceil \frac{10-2 \cdot 2}{2} \rceil - 1 = 2$ ), however also only RMW with  $d = 1$  is superior to FSW. Using Eq. 18, we get that for RAID-5 with  $n = 9$  RMW with  $d = 1$  surpasses FSW for logical requests up to 779.06 KiB and that for RAID-6 with  $n = 10$  RMW with  $d = 1$  surpasses FSW for logical requests up to 377.23 KiB. As a result, we consider logical request sizes of 256 KiB, 512 KiB, and 1024 KiB. We are interested in the aggregated random write throughput when a logical random write affects  $d \in \{1, 2, 4, 8\}$  whole data blocks. For both RAID setups, a logical random write will incur an RMW in case of  $d = 1$  and  $d = 2$ , an RCW for  $d = 4$  as well as an FSW if  $d = 8$ . In order to ensure that each logical write affects a particular number of whole data blocks we set the stripe unit size to  $u = s/d$  and align the starting address of each logical write to the boundaries of blocks comprising  $s$  KiB to ensure that each logical write stays within a single stripe (since otherwise a logical write may result in an unwanted code block update method, e.g., in two RMWs instead of a single RCW in case of  $d = 4$ ). The resulting aggregated random write throughput of a correspondingly configured HDD RAID-5 ( $k = 1$ ) as well as HDD RAID-6 ( $k = 2$ ) based on the average over 100,000 logical writes for each logical request size is shown in Fig. 9.

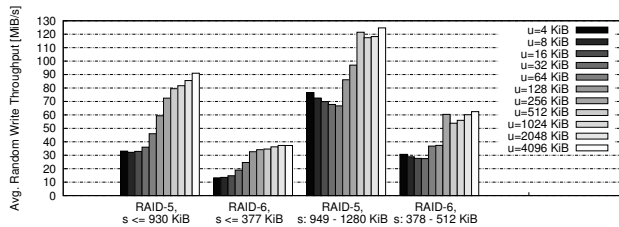
First, we take a look at the results for the HDD RAID-5 ( $k = 1$ ) with  $n = 9$ , where RMW with  $d = 1$  is superior to FSW up to a logical request size of 779.06 KiB. According to Fig. 9, RMW with  $d = 1$  yields the highest throughput for logical requests comprising 256 KiB and 512 KiB, while FSW results in the highest throughput for a logical request size of 1024 KiB. Next, we consider the HDD RAID-6 ( $k = 2$ ) with  $n = 10$ , where RMW with  $d = 1$  is superior to FSW up to a logical request size of 377.23 KiB. The mea-

sured values in Fig. 9 clearly show that RMW with  $d = 1$  achieves the highest throughput for logical requests comprising 256 KiB, while FSW is superior for larger logical requests, i.e., at a request size of 512 KiB and 1024 KiB. In summary, the measured aggregated throughput values shown in Fig. 9 support our claim that up to a certain logical request size which can be calculated for a particular setup, the highest aggregated random write throughput is achieved in case of RMW.

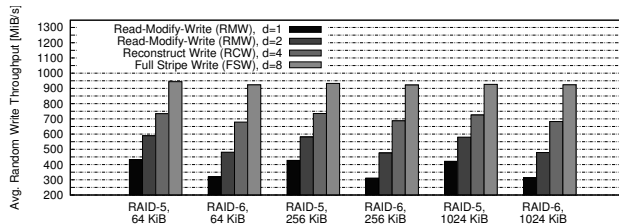
Now, we validate the recommendations for the choice of the stripe unit size in an erasure-coded HDD arrays (Sect. 4.1). This includes two claims: The first is that the aggregated random write throughput can be increased by tuning the stripe unit size such that logical requests whose request size lies in a range, where RMW is favorable over FSW end up in an RMW affecting the corresponding number of data blocks. The second is that if (almost) all logical requests have a request size larger than the logical request size where a RMW variant surpasses FSW, the aggregated random write throughput is potentially not improved by choosing the smallest possible stripe unit size in order to increase the number of FSWs (cf. Sect. 4.1).

To check this, we measured the average aggregated random write throughput of  $n = 10$  HDDs deployed once in a RAID-5 ( $k = 1$ ) and once in a RAID-6 ( $k = 2$ ) under random write workloads tailored to the request size range and for stripe unit sizes between 4 KiB (smallest possible value) and 4096 KiB (a multiple of the logical request size where RMW with  $d = 1$  surpasses FSW). Each workload consists of 100,000 writes with a randomly chosen starting address (aligned to sector boundaries due to direct I/O) and a randomly chosen request size  $s$  within a specific interval. In case of the HDD RAID-5, RMW with  $d = 1$  surpasses FSW up to a logical request size of 930.38 KiB and RMW with  $d = 2$  surpasses FSW and RMW with  $d = 1$  for logical requests between 895.92 KiB and 948.62 KiB. However, due to the small logical request size subinterval where RMW with  $d = 2$  is best and the inability of Linux Software RAID to use a stripe unit size of 896 KiB, we consider only the request size subinterval of RMW with  $d = 1$ . Thus, we measure in case of HDD RAID-5 the random write throughput once for logical requests between 512 bytes and 930 KiB and once for logical requests where FSW is superior, for example with logical request sizes between 949 KiB and 1,280 KiB. In case of HDD RAID-6, RMW with  $d = 1$  surpasses FSW and all RMW variants up to a logical request size of 377.23 KiB. Thus, we measure the random write throughput once for logical requests between 512 bytes and 377 KiB and once for logical requests where FSW surpasses RMW, namely for logical request sizes between 378 KiB and 512 KiB.

The results of the corresponding measurements are shown in Fig. 10. First, we consider the results for the two logical request size subintervals where RMW with  $d = 1$  is superior to FSW which are represented by the two leftmost groups of bars in Fig. 10. Looking at the results for both, HDD RAID-5 and HDD RAID-6, reveals that the highest throughput is achieved for the largest stripe unit size, which is 4096 KiB. This supports our claim that random write throughput can be increased in situations, where the size of (almost) all requests lies within a range, where RMW results in a higher



**Figure 10: Average random write throughput of a RAID-5 and a RAID-6 with 10 HDDs on varying stripe unit size  $u$  and logical request size  $s$ .**



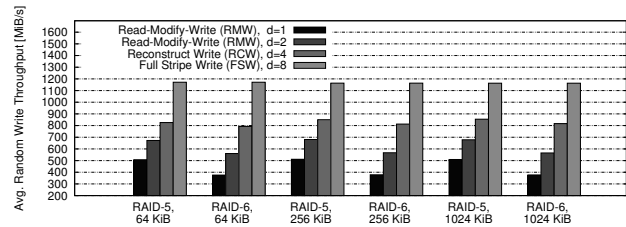
**Figure 11: Measured average random write throughput of Samsung 830 128 GB SSD RAID-5 and RAID-6 on varying code block update method and request size of logical writes.**

random write throughput than FSW, by tuning the stripe unit size such that logical writes end up in an RMW.

Next, we take a look at the results for the two logical request size subintervals where FSW is superior to all RMW variants, which are represented by the two rightmost groups of bars in Fig. 10. Obviously, setting the stripe unit size to the smallest possible value offered by Linux Software RAID (i.e., 4 KiB) in order to increase the number of FSWs does not yield the highest aggregated random write throughput. Instead, using a stripe unit size of 4096 KiB for the considered workloads is apparently a better choice for the HDD RAID-5 as well as the HDD RAID-6. This can be attributed to the fact that the logical writes do not end up only in FSWs, but also in RCWs and RMWs which decreases the throughput.

### 6.3.2 Random Write Throughput of SSD Arrays

Our analysis in Sect. 4.2 suggests that if random writes to an array with  $n$  SSDs which uses data striping with  $k$  erasure codes can be decomposed into  $\lceil \frac{n-2 \cdot k}{2} \rceil$  to  $n - k$  physical writes with a request size at which the used SSDs provide (nearly) the highest random write throughput, RCW and FSW result in higher aggregated random write throughput than RMW. We validate our analysis by measuring the random write throughput of preconditioned SSD-L with 27% OP deployed in a RAID-5 ( $k = 1$ ) and RAID-6 ( $k = 2$ ). Like for HDDs, we also consider setups with  $n - k = 8$ , i.e.,  $n = 9$  in case of RAID-5 ( $k = 1$ ) and  $n = 10$  in case of RAID-6 ( $k = 2$ ) in order to measure the aggregated random write throughput for each code block update method. We compare the aggregated random write throughput for a certain logical request size  $s$  under each code block update method in order to check whether RCW and FSW outperform RMW. We consider the case, where a logical write

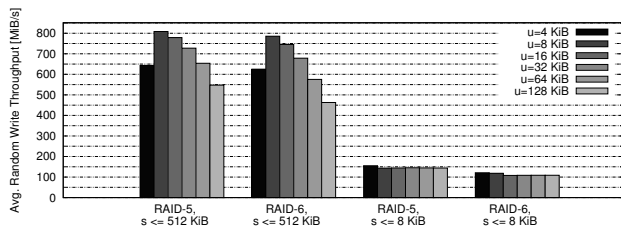


**Figure 12: Analytical average random write throughput of Samsung 830 256 GB SSD RAID-5 and RAID-6 on varying code block update method and request size of logical writes.**

affects  $d \in \{1, 2, 4, 8\}$  whole data blocks incurring either an RMW (if  $d = 1$  or  $d = 2$ ) or an RCW (if  $d = 4$ ), or an FSW (if  $d = 8$ ). We ensure that each logical write affects a particular number of whole data blocks in the same way as for HDDs, i.e., by setting  $u = s/d$  and by correspondingly aligning the starting address of each logical write. We consider three different logical request sizes which are chosen such that the resulting physical request size is an integral multiple of the flash page size (8 KiB) for any considered value of  $d$  and results in a wide range of physical request sizes (between 8 KiB and 1024 KiB): 64 KiB, 256 KiB, and 1024 KiB. This permits to study the influence of the (request size-sensitive) random read throughput to the aggregated random write throughput (of the SSD array) in case of RMW and RCW. However, we have not enough SSD-H to deploy the considered RAID-5/6 setups, hence, we provide analytical results based on our performance model in Sect. 3.2 and the measured throughput values of a single SSD-H 27% OP shown in Fig. 6. The measured values of the average aggregated random write throughput for SSD-L are shown in Fig. 11, while the analytical results are depicted in Fig. 12.

The measured aggregated random write throughput for SSD-L as shown in Fig. 11 confirms our analytical results (cf. Sect. 4.2) for logical writes comprising whole data blocks because RCW and especially FSW result in case of SSD RAID-5 as well as of SSD RAID-6 in a higher random write throughput than RMW. Moreover, considering setups with the same number of erasure codes  $k$  (i.e., RAID-5 with  $k = 1$  and RAID-6 with  $k = 2$ ) reveals that the aggregated random write throughput is nearly equally high for all three logical request sizes and, thus, virtually independent of the logical request size.

Looking at the analytical aggregated random write throughput values for SSD-H SSDs shown in Fig. 12 also indicates that the aggregated random write throughput is virtually independent of the logical request size, but the achieved write throughput values are slightly higher than that of SSD-L, which reflects the higher write throughput of a SSD-H (cf. Fig. 6(a)). For example, the throughput for FSWs amount approximately 1,200 MB/s in case of SSD-H (see Fig. 12), but only about 950 MB/s in case of SSD-L (see Fig. 11). However, for both versions of the Samsung 830 SSD, the throughput for RMW and RCW decreases with increasing number of erasure codes. This is reflected by a (generally) lower throughput in case of RAID-6 than in case of RAID-5 (for a comparable number of SSDs).



**Figure 13: Average random write throughput of Samsung 830 128 GB SSD RAID-5 with  $n=9$  and RAID-6 with  $n=10$  on varying stripe unit size  $u$ .**

After confirming our analysis of aggregated random write throughput in SSD arrays (using data striping with erasure coding) for requests affecting whole data blocks, we evaluate the benefit of device-aware data striping for workloads, where random writes can affect only a part of one or more data blocks (up to 2 data blocks) which results in sub-page (over-)writes. According to Sect. 4.2, the stripe unit size should be set to the flash page size. We evaluate this by measuring the aggregated random write throughput of SSD-L deployed in a RAID-5 setup with  $n = 9$  SSDs and in a RAID-6 setup with  $n = 10$  SSDs. In order to check if setting  $u = p = 8$  KiB (to the effective flash page size) is a good choice for mixed workloads (including sub-page and multi-page random writes) and workloads comprising only sub-page random writes, we measure the throughput for different stripe unit sizes.

We use a mixed workload that consists of 2,000,000 logical writes each having a randomly chosen starting address (aligned to sector boundaries) and a randomly chosen request size between 1 sector (512 bytes) and 1,024 sectors (512 KiB). Beside this, we apply a workload comprising (mostly) sub-page random writes with a randomly chosen request size between 1 sector (512 bytes) and 16 sectors (8 KiB) as well as with the same number of logical writes and starting address properties. We conduct random write throughput measurements for both workloads and striping unit sizes of 4 KiB (lowest possible value), 8 KiB, 16 KiB, 32 KiB, 64 KiB, and 128 KiB. Please note that for  $u = 128$  KiB no FSWs will occur since each stripe comprises 1024 KiB and the logical request size is at most 512 KiB. This permits to study the throughput in absence of FSWs, which should result in notably lower random write throughput compared to smaller stripe units. The measured aggregated random write throughput is depicted in Fig. 13.

First, we consider the results for the mixed workload (comprising sub-page and multi-page random writes) that are represented by the two leftmost groups of bars in Fig. 13. Obviously, the highest aggregated random write throughput is achieved when the stripe unit size is set to the capacity of exactly one flash page, i.e., 8 KiB, which coincides with our argumentation in Sect. 4.2. Moreover, the lowest throughput occurs if the stripe unit size is chosen such large that no FSWs occur, which is the case for  $u = 128$  KiB. Now, we take a look at the results for the sub-page random writes represented by the two rightmost groups of bars in Fig. 13. In case of RAID-5 the attained aggregated random write throughput is nearly equally high for all stripe

unit sizes except of 4 KiB (which is about 10% higher). In case of RAID-6, the highest aggregated random write throughput is reached using a stripe unit size of 4 KiB and 8 KiB, while the throughput is comparably high for larger stripe unit sizes. The fact that a slightly higher throughput (about 10%) is achieved for the smallest (compared to larger) stripe unit sizes stands to some extent in contrast to our argumentation that sub-page writes should affect only one data block to prevent that they end up in multiple even smaller physical requests. It would be interesting to check if the trend towards higher throughput at even lower stripe unit sizes (i.e., less than 4 KiB) continues, but the smallest possible the stripe unit size is 4 KiB when using Linux Software RAID.

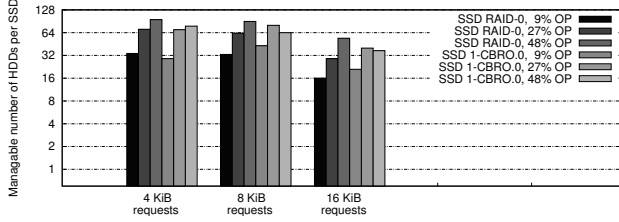
## 6.4 Code Block Request Offloading

In the following, we present an experimental evaluation of the benefits of CBRO. We use HDDs as data drives and SSD arrays as CDAs. In contrast to our previous work [12] we also consider larger request sizes due to the fact that in HDD arrays (using data striping with erasure coding) RMWs can be favorable for random write sizes in the range of several hundreds of KiB (depending on the used HDD model and the data striping configuration). First, we assess the CDA scalability. Afterwards, we experimentally evaluate the results of our write performance analysis in Sect. 5.2 and we examine the SSD write endurance in CBRO setups. However, as a consequence of our findings in Sect. 6.3, we use a device-aware data striping configuration for HDD and SSD arrays that employ data striping with erasure coding.

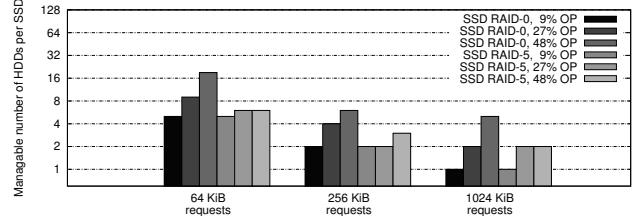
### 6.4.1 Check Drive Array Scalability

The first step is to determine the required number of drives in each CDA (cf. Sect. 5.3). In this evaluation, 1 TB HDDs are used as data drives, and SSDs with capacities of 128 GB (SSD-L) as well as 256 GB (SSD-H) are used in CDAs. Since we intend to use the whole storage capacity of each HDD, each CDA must comprise at least 8 SSD-L or 4 SSD-H to provide enough capacity for all code blocks related to an erasure code. However, as we have 7 SSD-H at our disposal, we consider CDA setups with up to 7 SSD-H and 14 SSD-L. In order to assess the necessary CDA performance to cope with all code block requests from up to 16 HDDs, we measure the average random RMW throughput of an HDD and the considered SSD-based CDAs on varying request size. Please note that we use disk form factor SSDs whose random access performance is limited due to the usage of standard disk host interface (SATA-3 with a maximum throughput of 600 MB/s). Due to this, with increasing random RMW request size the throughput advantage of SSDs over HDDs is shrinking (see Sect. 5.4.1).

In order to determine the scalability of a particular CDA setup, we consider the number of HDDs whose requests can be served by a single SSD in a certain CDA setup. Consequently, we measured the average random RMW throughput of a single HDD and of CDA setups for request sizes of 4 KiB, 8 KiB, 16 KiB, 64 KiB, 256 KiB, and 1024 KiB. In the following, we differentiate between small requests (i.e., 4 KiB, 8 KiB, and 16 KiB) and larger requests (i.e., 64 KiB, 256 KiB, and 1024 KiB). As a consequence of our results in



(a) Small Request Size



(b) Larger Request Size

**Figure 14: Scalability of different CDA setups on varying request size and degree of OP with respect to the number of HDDs whose code block requests can be served by a single SSD (base-2 logarithmic scaled y-axis).**

Sect. 6.3, we apply device-aware data striping configurations to CDAs that involve erasure coding: For small requests up to 16 KiB (two flash pages), RMW is favorable over RCW for  $n \geq 8$  and we use nested CBRO setups as CDAs like in our previous work [12]. However, for larger requests RCW and FSW are favorable over RMW, thus we use SSD RAID-5 CDAs because CBRO provides almost no benefit in case of RCW and FSW (Sect. 5). Please note that in case of a CDA based on a nested CBRO setup, we deployed 4 SSD-L in the SSD RAID-0 sub-CDA in case of 9% OP (8 SSD-L as data drives) and, for larger degree of OP, we deployed 8 SSD-L in the SSD RAID-0 sub-CDA, because an SSD RAID-0 sub-CDA with 4 SSD-L cannot cope with the RMW requests from 10 and 14 SSD-L (data drives). In summary, the considered CDAs based on a nested CBRO setup are 1-CBRO.1-CBRO.0-8L-4L in case of 9% OP, 1-CBRO.1-CBRO.0-10L-8L in case of 27% OP, and 1-CBRO.1-CBRO.0-14L-8L in case of 48% OP (cf. Sect.6.1). Based on the measured average RMW throughput values for each CDA setup and a single HDD, we calculated the number of HDDs whose code block requests can be served by a single SSD and a single data SSD in a nested CBRO setup, respectively. The corresponding scalability results with regard to RMW request size and OP are shown in Fig. 14.

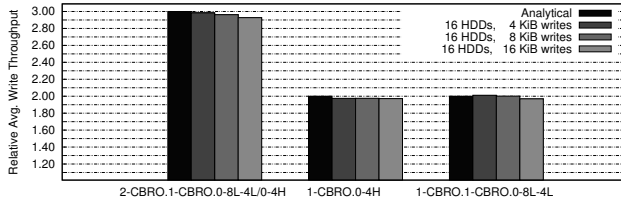
First, we take a look at the results for small writes shown in Fig. 14(a). It can be noticed that an SSD in an RAID-0 CDA (3 leftmost bars in each group) is able to cope with more data HDDs than an 1-CBRO.0 CDA. Considering SSD RAID-0 CDAs, the scalability noticeably increases with the degree of OP for all request sizes. However, considering SSD 1-CBRO.0 CDAs, this does not apply for requests comprising 8 or 16 KiB and the highest degree of OP (48%), represented by the rightmost bars in each group, because the number of HDDs whose code block requests can be served by a single SSD decreases (instead of increasing). The reason is the saturation of the SSD RAID-0 sub-CDA comprising 8 SSD-L. In order to overcome this, either even more SSD-L or faster SSDs are necessary. Apart from this, a comparison of the results for the three request sizes shows that the results for 4 or 8 KiB are similar, while for 16 KiB, each SSD can cope with the load of less HDDs. However, even with the lowest degree of OP, and hence, lowest increase of hardware costs, a single SSD can cope with the code block requests from 16 HDDs in case of 16 KiB requests and 32 HDDs for 4 and 8 KiB requests. This makes CBRO appealing for workloads dominated by small random writes.

When considering the results for larger writes as depicted in Fig. 14(b), it can be noticed that a single SSD in a CDA can in general cope only with the code block requests of notably less HDDs than for small requests. This is attributed to the limited random access throughput of the used disk form factor SSDs, since the random access throughput of the HDDs increases with growing request size, while the random access throughput of the SSDs is limited by the used disk host interface. However, like for small requests, the scalability increases with the degree of OP, especially when comparing the lowest OP (9%) with a reasonable degree of OP (27%). Moreover, CDAs based on SSD RAID-0 provide better scalability than CDAs based on SSD RAID-5. This stems from additional reads/writes that are necessary to keep the code blocks up to date, thus, reducing the aggregated RMW throughput of a CDA based on an SSD RAID-5.

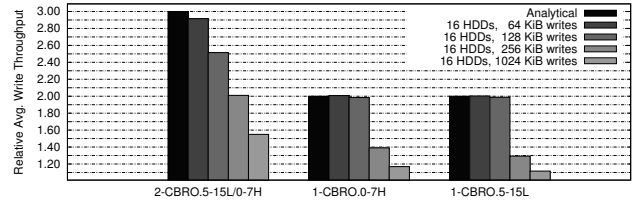
#### 6.4.2 Random Write Throughput

According to our analysis in Sect. 5.2, applying CBRO improves the throughput of random writes in case of an RMW compared to setups with distributed code blocks. We extend our previous work on CBRO [12] by focusing on the throughput for random single block writes with notably larger block sizes (up to 1,024 KiB). In order to investigate the performance improvement due to the use of CBRO, we have conducted random write throughput measurements for different CBRO setups using 16 HDDs as data drives as well as for HDD RAID-5 and HDD RAID-6, both with 16 HDDs. Since we are interested in the throughput increase due to CBRO, we report the relative throughput compared to RAID setups with the same number of erasure codes  $k$ , but distributed code blocks.

Consequently, the average random write throughput of 1-CBRO setups is compared to RAID-5, and of 2-CBRO setups to RAID-6. The measurements are based on 500,000 random writes per repetition and for each considered (logical) request size. In case of small random writes comprising 4, 8 or 16 KiB, the measured values were virtually independent of the considered degrees of OP, consequently we report only the values for the lowest (9%) degree of OP. However, in contrast to this, for larger random writes comprising 64, 256 or 1024 KiB, we report the values in case of CDA setups with the highest number of SSDs and degree of OP (48%), because for 1024 KiB random writes SSD RAID-0 based CDA is not fast enough to cope with the RMW requests from 16 HDDs.



(a) Small Request Size



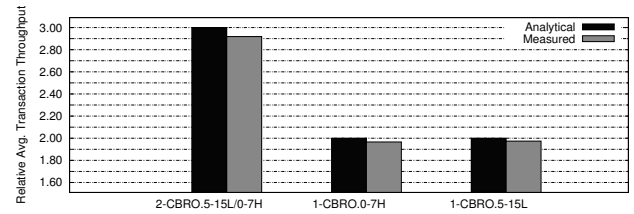
(b) Larger Request Size

**Figure 15: Average throughput of random single block writes with varying request size in different CBRO setups relative to pure HDD RAID-5/6.**

The relative throughput of random single block writes is shown in Fig. 15. First, we look at the results for small random writes in Fig. 15(a). The relative throughput of each CBRO setup is very close to the analytical speedup values, which are 2 in case of one erasure code (1-CBRO) and 3 for two erasure codes (2-CBRO). Next, we consider the results for larger random writes shown in Fig. 15(b). Looking at the results reveals that for a request size of 64 KiB the throughput of each CBRO setup is very close to the analytical speedup value, which is 2 in case of one erasure code (1-CBRO) and 3 for two erasure codes (2-CBRO). However, for random writes comprising 256 KiB and 1,024 KiB the analytical speedup values were not reached despite the fact that we used CDA setups with the highest number of SSDs and degree of OP (48%), which are according to our scalability results in Sect. 6.4.1 able to cope with the RMW requests from 16 HDDs (cf. Fig. 14(b)). In order to examine the reduction of the relative throughput for random writes above 64 KiB in greater detail, we also conducted measurements for 128 KiB. Looking at the results for 128 KiB in Fig. 15(b) indicates that the speedup values in case of 1-CBRO setups coincide with the analysis, while in case of the 2-CBRO setup the speedup is only 2.5 instead of 3.0.

In order to find the cause for the throughput discrepancy, we tracked the utilization of the HDDs (data drives) and all drives within the CDAs, which is reported by the *fio* benchmark averaged over the whole throughput measurement duration and based on the I/O statistics reported by the Linux kernel. The utilization values indicate that neither the SSDs deployed CDA nor the HDDs were saturated, thus, the throughput is not limited by the HDDs or SSDs. Next, we examined the CPU utilization of the tasks related to the *fio* benchmark and *MD RAID 4/5/6* driver. None of the tasks belonging to the *fio* benchmark saturated a CPU core, thus, we exclude *fio* as reason. Contrarily, in case of the 2-CBRO the corresponding RAID driver task saturated a CPU core for 128 KiB requests representing a bottleneck.

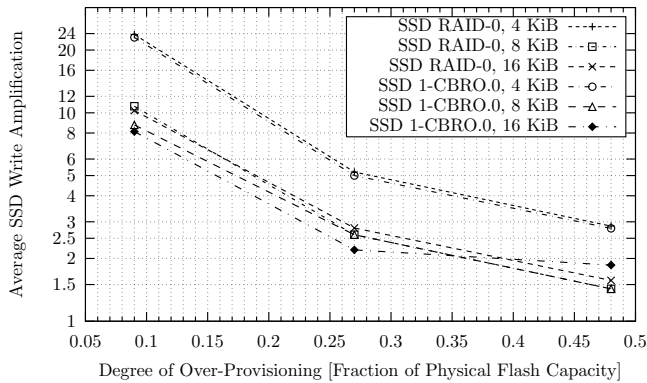
Distributing the processing of random writes to multiple threads (Sect. 6.1) on different CPU cores did not result in a higher throughput. Similarly, for random writes comprising 256 KiB and 1,024 KiB, using multiple threads (scheduled to different CPU cores) for random write processing also did not help increasing the throughput. However, this can be partially attributed to the maximum stripe buffer size of 32,768 (which is the number of 4 KiB memory pages) that is reached at a request size of 256 KiB (corresponding



**Figure 16: Average random write throughput of different CBRO setups relative to pure HDD RAID-5/6 for requests between 512 bytes and 128 KiB.**

to 64 memory pages) in combination with a queue depth of 512. Nevertheless, despite the insufficient stripe buffer size, the *MD RAID 4/5/6* driver apparently suffers from a write performance issue for larger requests that deserves further investigation because this prevents reaching the predicted random write throughput of CBRO.

After investigating the throughput increase (due to CBRO) for random writes of particular request size while writing whole data blocks (which was also the case in our previous work on CBRO [12]), we examine the benefit of CBRO for workloads that comprise random writes of different sizes and, where a logical random write can also affect data blocks only partially. For this purpose we use a mixed workload that consists of 500,000 logical writes each having a randomly chosen starting address (aligned to sector boundaries) and a randomly chosen request size between 1 sector (512 bytes) and 256 sectors (128 KiB). Please note that we do not consider larger requests due to the performance issues with the *MD RAID 4/5/6* driver. However, in order to achieve that logical writes end up in an RMW (otherwise CBRO would yield very little benefit compared to an RMW), we set the stripe unit size for the data HDDs to 1,024 KiB making an RCW quite unlikely because RMWs occur for logical writes that affect up to  $\lceil \frac{18-2.1}{2} \rceil - 1 = 7$  in case of  $k = 1$  and up to  $\lceil \frac{18-2.2}{2} \rceil - 1 = 6$  data blocks within a stripe in case of  $k = 2$ , respectively. Looking at the analytical and measured results shown in Fig. 16 indicates that the measured random write throughput coincides with our analysis. As a consequence, heterogeneous drive arrays comprising HDDs as data drives and SSDs as CDAs benefit from CBRO also when logical random writes cover data blocks only partially but as long as they end up in an RMW.



**Figure 17: Average SSD write amplification in CDA setups for small writes on varying over-provisioning.**

### 6.4.3 SSD Write Endurance

The SSD write endurance has a large impact on the hardware costs of a CBRO setup and, thus, its cost-benefit ratio. However, the write endurance is determined by the technological properties of the used flash memory, and the WA. In our testbed, the technological properties are laid down by the used SSD model. As a result, the write endurance will be primarily determined by the amount of WA which depends very much on the degree of OP [7, 11], but also on the write request size. The dependency of the WA to the write request size stems from the fact that larger requests will invalidate more flash pages at a time, hence, the GC has to relocate less flash pages with valid data before block erasure, resulting in a lower WA.

In order to assess the SSD write endurance, we have determined the WA experimentally depending on the degree of OP. For this, the amount of P/E cycles performed by each SSD (SMART attribute *177 Wear Leveling Count* [22, Ch. 7]) was compared before and after performing an amount of RMW requests (with a particular size) to a CDA that corresponds to the size of its physical flash capacity. However, since random writes of 64 KiB, 256 KiB, and 1024 KiB impose only a barely measurable WA of up to 0.15 (64 KiB random writes), we focus in the following on small random writes, i.e., 4 KiB, 8 KiB, and 16 KiB. The resulting WA values (averaged over all data SSDs in a CDA) for small random writes are depicted in Fig. 17.

The results in Fig. 17 show that the average WA for 4 KiB requests is about twice as high as for 8 KiB and 16 KiB requests, regardless of the degree of OP. This is attributed to the fact that the used Samsung 830 SSDs have an effective flash page size of 8 KiB. As a result, a random write of 4 KiB incurs writing a whole flash page comprising 8 KiB doubling the WA. Additionally, the results indicate that WA is roughly inversely proportional to the degree of OP, whereby the proportionality factor depends on request size and CDA setup. For example, with 9% OP the WA amounts 23–24 for 4 KiB requests and 8–11 for 8 KiB/16 KiB requests, respectively. Increasing the degree of OP to 27% results in a WA of about 5 for 4 KiB requests and 2–3 for 8 KiB/16 KiB requests, respectively. Furthermore, by increasing the degree of OP to 40% (cf. Fig. 17), the WA can be reduced

below 2 for 8 KiB/16 KiB requests. Similar to the benefit of a reasonable degree of OP (27%) for CDA performance (cf. Sect. 6.4.1), a significant SSD write endurance increase can be achieved at acceptable expense (adding one SSD for each four in a CDA). The write endurance per SSD in TBW (terabytes written) results from the drive capacity, measured WA, and manufacturer's endurance rating of about 3,500 P/E cycles (derived from the normalized and raw value of the SMART attribute). As a result, in case of 4 KiB random writes and 27% OP (WA of approx. 5) a single SSD yields at least 89.6 TBW (SSD-L) and 172.3 TBW (SSD-H), respectively. This corresponds to about 700 times of the nominal capacity. However, for mixed workloads comprising also larger random writes, the TBW values will be notably higher and the SSDs can be used for a longer time. We illustrate the effect of small (sub-page) random writes on the write endurance by the following example.

We consider the write endurance of a fully utilized (in terms of requests) 1-CBRO.0-5H setup with 16 HDDs (1,600 IOPS) in face of sub-page random writes comprising 4 KiB, which represents a worse case due to the high WA. A 1-CBRO.0-5H setup implies a SSD RAID-0 CDA with 5 SSD-H and 27% OP. According to our results, the WA is 5.2 in case of 4 KiB requests. Now we can calculate the time that is necessary to exhaust the P/E cycles of all SSDs for 4 KiB requests. Each of the 5 SSD-H comprises (with 27% OP) 67,108,864 of 4 KiB blocks. Thus, exhausting the P/E cycles of all SSDs would take  $\frac{(3,500/5.2) \cdot 67,108,864}{1,600/5} \approx 141,154,462$  seconds, which is about 1,622 days or around 4.44 years. Please note that when the P/E cycles of all SSDs are used up, the redundancy provided by the code blocks is lost and not data. However, in order to re-establish the redundancy all SSDs in the CDA must be replaced.

## 7. RELATED WORK

The following overview about related approaches is split into two parts that correspond to the main contributions of this article. First, we discuss related work that aims at improving the random write performance of homogeneous RAID arrays solely consisting of either HDDs or SSDs. Thereafter, we survey those approaches that combine HDDs and SSDs in heterogeneous drive arrays in order to combine the advantages of both storage technologies.

**Device-Aware Data Striping.** The choice of the stripe unit size in HDD arrays using data striping with erasure coding was studied by Chen and Lee [3]. They developed a method to determine the most appropriate stripe unit size for an HDD RAID-5 with respect to the performance characteristics of the used HDD model, i.e., the average head positioning time and average data transfer rate. In contrast to our results for HDDs in Sect. 4.1, Chen and Lee did not conclude that in larger arrays RMW yields a notably higher random write throughput than FSWs up to a particular logical random write request size. Besides, Chen and Lee did not consider HDD arrays using more than one erasure code.

Salmasi et al. [20] examined the impact of the stripe unit size on the write endurance and performance of SSD RAID5 using data striping with one erasure code (i.e., parity). Based

on an analysis of the number of flash pages that have to be read and written in the event of a logical write depending on the used stripe unit size, Salmasi et al. conclude that setting the stripe unit size to the flash page size results in the lowest number of flash page reads and writes, thus, in the highest write performance in case of an erasure-coded SSD array. However, while our analysis for SSDs in Sect. 4.2 leads to the same conclusion, Salmasi et al. did not consider the case that logical writes can have a request size smaller than the flash page size (i.e., sub-page writes). Thus, our analysis extends the work of Salmasi et al. in this regard.

**Code Block Request Offloading.** Wacha et al. [31] first proposed to store code blocks on much faster devices (i.e., SSDs). They showed that write penalties are not only mitigated, but on the contrary, considerable performance improvements are achieved especially for small to medium-sized random writes. However, as their *RAID4S* approach stores all code blocks on a single SSD (cf. [30, Sect. 3.8]), the usable space on each HDD within the setup is restricted to the SSD's capacity which is often much smaller. Hence, RAID4S setups tend to be uneconomical and of limited use for many practical applications.

The *Splitting Parity Disk-RAID4 (SPD-RAID4)* architecture by Pan et al. [19] uses a single erasure code while storing its code blocks onto a dedicated SSD RAID-0 array and, thereby, allows to use larger capacities. However, the authors focused only on homogeneous SSD setups while aiming to improve the random write performance by separating requests to code and data blocks. Furthermore, they neglected reliability aspects albeit using a potentially large check drive array without any kind of fault tolerance mechanism.

In our previous work on CBRO [12], we considered both capacity and reliability by using SSD-based check drive arrays of arbitrary size for multiple erasure codes and by recursively applying CBRO to check drive arrays in order to establish fault tolerance when a larger number of drives is used. However, we focused on scenarios, where small random writes comprising a few KiB are dominant. Due to the insight that the random write throughput of erasure-coded HDD arrays can be higher in case of RMW compared to RCW and FSW (depending on the used HDD model and the array configuration), CBRO is also beneficial for random writes comprising hundreds of KiBs, when using HDDs as data drives. Hence, this article extends our previous work on CBRO by taking a broader range of workloads into account.

## 8. CONCLUSIONS AND FUTURE WORK

Random writes are challenging for most data storage systems, especially, if erasure coding is used to ensure the reliability of the stored data and to maintain availability in the event of drive failure(s). In this article, we focused on improving the random write throughput in such data storage systems. We provided an analytical model to predict the random write throughput of homogeneous erasure-coded drive arrays, comprising either conventional HDDs or NAND flash memory SSDs, based on the random write throughput of a single drive and if a write affects a certain number of data blocks within a stripe. Based on this model, we de-

scribed a method (denoted as device-aware data striping) to improve the random write throughput of HDD-only and SSD-only arrays, which takes the notably different random access performance characteristics of HDDs and SSDs into account. This is achieved by adapting the stripe unit size to the used device type and model in relation to the workload characteristics. While our recommendations for the stripe unit size in case of HDDs are limited to certain workload conditions, this does not apply for our recommendations regarding SSDs. However, our experimental evaluation confirmed our analysis and the benefits of the proposed recommendations for the stripe unit size.

In addition to this, we extended our previous work on an organization for heterogeneous drive arrays referred to as CBRO, where data blocks are accommodated by slower and code blocks by faster drives, by applying our results for homogeneous erasure-coded drive arrays to CBRO and by considering also workloads that are not limited to small random writes of only a few KiB. Our experimental results show that CBRO is especially suitable for arrays combining HDDs with SSDs for small as well as larger random writes that comprise up to hundreds of KiBs.

We plan to extend our analysis as well as our recommendations for the stripe unit size in case of HDDs in order to cover a broader range of workloads. Besides, based on our results for homogeneous and heterogeneous erasure-coded drive arrays, we want to develop mechanisms that try to dynamically adapt the data organization to the workload in order to increase the overall system performance.

## 9. REFERENCES

- [1] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2009)*, pages 181–192, New York, NY, USA, 2009. ACM.
- [2] F. Chen, R. Lee, and X. Zhang. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA 2011)*, pages 266–277, Feb 2011.
- [3] P. M. Chen and E. K. Lee. Striping in a RAID level 5 disk array. In *Proceedings of the ACM Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 1995)*, pages 136–145, New York, NY, USA, 1995. ACM.
- [4] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: high-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2):145–185, 1994.
- [5] HGST, Inc. *Ultrastar 15K600 3.5 inch Serial Attached SCSI (SAS) Hard Disk Drive*, Sept. 2012. [https://www.hgst.com/sites/default/files/resources/US15K600\\_SAS\\_Spec\\_V2.00.pdf](https://www.hgst.com/sites/default/files/resources/US15K600_SAS_Spec_V2.00.pdf).
- [6] W. W. Hsu and A. J. Smith. The real effect of I/O optimizations and disk improvements. Technical

- Report UCB/CSD-03-1263, EECS Department, University of California, Berkeley, CA, USA, Jul 2003.
- [7] X.-Y. Hu, E. Eleftheriou, R. Haas, I. Iliadis, and R. Pletka. Write amplification analysis in flash-based solid state drives. In *Proceedings of the Israeli Experimental Systems Conference (SYSTOR 2009)*, pages 10:1–10:9, New York, NY, USA, 2009. ACM.
  - [8] Hynix Semiconductor Inc. *16Gb NAND FLASH HY27UH08AG5B Data Sheet*, Jan. 2008. Rev. 0.2.
  - [9] Intel Corporation. *Intel MD332 NAND Flash Memory Data Sheet*, June 2009. Document Number 319120-004US.
  - [10] B. L. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.
  - [11] N. Jeremic, G. Mühl, A. Busse, and J. Richling. The pitfalls of deploying solid-state drive raids. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR 2011)*, pages 14:1–14:13, New York, NY, USA, June 2011. ACM.
  - [12] N. Jeremic, H. Parzyjegl, and G. Mühl. Improving random write performance in heterogeneous erasure-coded drive arrays by offloading code block requests. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing (SAC 2015)*, pages 2007–2014, New York, NY, USA, 2015. ACM.
  - [13] M. Jung and M. Kandemir. Revisiting widely held SSD expectations and rethinking system-level implications. In *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2013)*, pages 203–216, New York, NY, USA, 2013. ACM.
  - [14] R. Micheloni, A. Marelli, and K. Eshghi. *Inside solid state drives (SSDs)*, volume 37 of *Springer Series in Advanced Microelectronics*. Springer, Dordrecht, Netherlands, 2012.
  - [15] Micron Technology Inc. *MT29F16G08ABACA Data Sheet*, Sept. 2012. Rev. F.
  - [16] Micron Technology Inc. *MT29F64G08CBAA Data Sheet*, May 2012. Rev. F.
  - [17] S. Moon and A. L. N. Reddy. Don’t let RAID raid the lifetime of your SSD array. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems (HotStorage 2013)*, Berkeley, CA, USA, 2013. USENIX Association.
  - [18] NVM Express Workgroup. NVM Express 1.2a, Oct. 2015.
  - [19] W. Pan, F. Liu, T. Xie, Y. Gao, Y. Ouyang, and T. Chen. SPD-RAID4: splitting parity disk for RAID4 structured parallel SSD arrays. In *Proceedings of the 15th IEEE International Conference on High Performance Computing and Communications (HPCC 2013)*, pages 9–16, Los Alamitos, CA, USA, Nov. 2013. IEEE Computer Society.
  - [20] F. R. Salmasi, H. Asadi, and M. GhasemiGol. Impact of stripe unit size on performance and endurance of SSD-based RAID arrays. *Scientia Iranica – Transactions D: Computer Science & Engineering and Electrical Engineering*, 20(6):1978–1998, May 2013.
  - [21] Samsung Electronics Co., Ltd. *K9ACGD8X0A Data Sheet*, Oct. 2011. Rev. 0.1.
  - [22] Samsung Electronics Co., Ltd. *Samsung Solid State Drive – White Paper*, 2013.
  - [23] S. Sankar, S. Gurumurthi, and M. R. Stan. Intra-disk parallelism: An idea whose time has come. In *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA 2008)*, pages 303–314, Washington, DC, USA, 2008. IEEE Computer Society.
  - [24] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *In Proceedings of the USENIX Winter Technical Conference (USENIX Winter 1990)*, pages 313–324, Berkeley, CA, USA, 1990. USENIX Association.
  - [25] Serial ATA International Organization. Serial ATA Revision 3.2, Aug. 2013.
  - [26] Small Form Factor Committee. *Self-monitoring, analysis and reporting technology (S.M.A.R.T.)*, revision 2.0 edition, Apr. 1996.
  - [27] Technical Committee T10. SCSI Block Commands - 3 (SBC-3), Oct. 2011.
  - [28] A. Thomasian. Reconstruct versus read-modify writes in RAID. *Information Processing Letters*, 93(4):163–168, Feb. 2005.
  - [29] Toshiba Corporation. *TC58NVG6T2FTA00 Data Sheet*, Dec. 2010.
  - [30] R. Wacha. *Incorporating solid state drives into distributed storage systems*. PhD thesis, University of California at Santa Cruz, Santa Cruz, CA, USA, Dec. 2012. <http://escholarship.org/uc/item/3b37r7f4>.
  - [31] R. Wacha, S. Brandt, J. Bent, and C. Maltzahn. RAID4S: Adding SSDs to RAID arrays. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST 2010), Poster Session*, Berkeley, CA, USA, Feb. 2010. USENIX Association.

## ABOUT THE AUTHORS:



Nikolaus Jeremic is a PhD student at the Architecture of Application Systems group at University of Rostock since 2010. He studied computer science at Goethe-University in Frankfurt am Main. In 2009, he received his diploma degree. His research focuses on adaptive data storage systems based on heterogeneous storage devices.



Helge Parzyjegl is a postdoctoral researcher at the Architecture of Application Systems group at University of Rostock. He studied computer science at the Berlin University of Technology. In 2013, he received a PhD in computer science from the University of Rostock. His research interests lie in the area of distributed, event-driven systems, publish/subscribe middleware, and storage architectures with the focus on aspects of self-organization and adaptive behavior.



Gero Mühl has studied computer science and electrical engineering at University of Hagen. In 2002, he received a PhD in Computer Science from Darmstadt University of Technology. From 2002 to 2009 he was a research assistant at the Communication- and Operating Systems group at Berlin University of Technology, where he was awarded with the habilitation in 2007. Since 2009 he is head of Architecture of Application Systems group at University of Rostock. His research interests lie in the fields of self-organizing distributed systems, event-driven architectures, and adaptive storage systems.

# Optimizing Swap Space for Improving Process Response after System Resume

Shiwu Lo, Hung-Yi Lin, Zhengyuan Chen

Computer Sciences and Information Engineering

National Chung-Cheng University

168, University Rd., Min-Suing, Chia-Yi, Taiwan, R.O.C.

+886-52320411

{shiwulo, lhyi100m, ccyan100m}@cs.ccu.edu.tw

## ABSTRACT

The swap-before-hibernate (SBH) swaps out all swappable pages before the system enters hibernation mode, thus reducing the memory used by the system and the size of hibernation file. However, when the system resumes, the program must be reloaded into the main memory by swap-in; hence, the response time of program depends on the swap-in speed.

This study experimentally proved that the present swap-in/swap-out mechanism of Linux is not acceptable to the SBH algorithm. Moreover, it proposed the method of reordering all the requests to be written out to the swap space. The proposed method was proven to increase the efficiency of swap space by 1/2 to 1/3.

## CCS Concepts

• Computer systems organization~Embedded software • Software and its engineering~Operating systems

## Keywords

Quick booting; Android; Linux; Hibernation; Suspend to Drive

## 1. INTRODUCTION

Under the advancement of science and technology, flash memory has become the main-stream storage tool in embedded systems. As compared to the traditional storage technologies, such as hard disk, flash memory is featured by small size, shock-proof, low power consumption, and compatibility with smart devices, such as digital household appliances, and automotive electronics [1, 2]. The cost performance of flash memory has been improved [3]. Although the access time of flash memory has made great progress, smart embedded systems, such as Android and iOS, have become increasingly complex and main memory has become larger, thus extending the booting time. For smart TV users, the booting time of 30 seconds is too long [4]. For automobile drivers, they expect to use back-up collision intervention system and car navigation system immediately when they ignite cars. Manufacturers often use suspend-to-RAM (sleep) [5], which is power-consuming standby, to achieve quick booting time to meet the user demands. Moreover, mobile embedded systems require

greater battery storage, and smart household appliances cannot meet the requirement of green energy. Another commonly used technology is suspend-to-drive (hibernation), which can accelerate booting time and achieve zero standby power consumption. As compared to suspend to RAM, the booting time of suspend-to-hard-drive is longer.

Hibernation (suspend-to-drive) [5] refers to a system status during operation, including hardware environment state, memory content and processor status, which is stored in hibernation file in the non-volatile storage device. The device is powered down to achieve zero power consumption. After the system is restarted, the hardware device reads hibernation file back to main memory, and restores the system to work state before hibernation. The system recovery time depends on the size of hibernation file. When more system memory is used before hibernation, the recovery time is longer. Lo et al. [6] proposed Swap-before-hibernate (SBH), which uses flash memory as system's secondary storage. They found that the random access time of the flash memory is close to sequential access time and the fast random access could reduce response time significantly by adopting SBH.

In SBH, when entering the hibernation mode, the swappable memory swap-out to the swap space or the file system. The least amount of data was stored in hibernation file. During booting, a smaller hibernation file is read back through sequential access of the flash memory. After the booting is completed, the system swaps in data quickly through superior random access of the flash memory, which is closed to the sequential access speed. Thus, the booting time depends on the size of hibernation file in sequential access, and the response time of a program depends on the speed/quantity of swap-in pages in random access. In summary, SBH recovery is divided into two stages. The first stage is to read hibernation file in the flash memory. After loading hibernation file, OS kernel starts to load memory pages of application programs by swapping. In loading application programs, OS kernel can find that some accessed data are not stored in the main memory. This is because that most of data have been swapped out to swap space or file system before the system entering hibernation mode, thus the system uses swap-in to load data from flash memory.

This study aims to reorganize/reschedule the data that swap out to swap space to shorten the response time of applications after resuming from hibernation. Please notice that the OS kernel swaps out memory pages individually, while the main memory is

Copyright is held by the authors. This work is based on an earlier work: RACS'15 Proceedings of the 2015 ACM Research in Adaptive and Convergent Systems, Copyright 2015 ACM 978-1-4503-3738-0. <http://dx.doi.org/10.1145/2811411.2811464>

insufficient, but the OS kernel generally does not swap out all swappable memory. In normal case, as OS kernel only swaps out some pages at one time, and only the greedy algorithm can be used for optimization. In SBH algorithm, as OS kernel swaps out all the memory pages of all processes to the swap space at one time, we can reorder swap-out commands according to the characteristics of memory pages, so as to optimize the swap space.

The rest of the paper is organized as follows. In the second section we show the motivation. In section 3 we discuss related works. The method to enhance performance of swap space is discussed in section 4. Section 5 gives experimental results. Section 6 is conclusions.

## 2. MOTIVATION

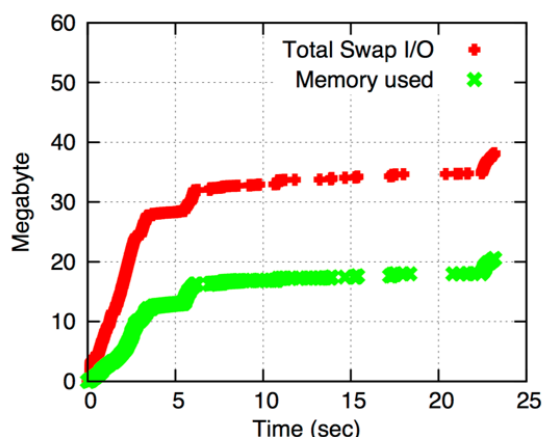
Most modern operating systems, such as Android/Linux, GNU/Linux, UNIX and Windows, use memory management unit (MMU) to manage memory. The MMU divides main memory into pages. In most case, the memory pages have the same size (e.g., 4KB). The operating systems divide these pages into several categories: (1) Kernel pages: these pages are owned by OS kernel, and are always stored in the main memory; (2) named pages: these pages are read from a file and the backing store of the page is the said file; if the page contents are modified, the OS is necessary to maintain consistency between the page and the backing store; (3) anonymous pages: anonymous pages are so named, because they have no backing store (a named file) in the file system; most programs request an anonymous page to store dynamically allocated data (e.g., stack and heap); (4) page cache: for the purpose of efficiency, the operating system reads several blocks (e.g., 8 blocks at a time) from file system at a time to reduce the number of I/O, and these blocks are stored in page cache. When a program accesses a block in a secondary storage, OS would first check whether the block is in the page cache. If it is in the page cache, OS can return the block to the program without I/O.

When the OS swaps out dirty pages, the data are written in the secondary storage to keep consistency. OS adopts different mechanisms according to the types of swap-out page. For four different pages, the mechanisms adopted by OS are described below: (1) kernel pages: OS never swap out kernel pages; (2) named pages: the pages are directly swapped out without I/O if these pages are clean pages; if dirty pages are found, release is made after these file pages are written back in the file system; (3) anonymous pages: except for dirty pages written in swap-space, the other steps are identical with named pages; (4) page cache: the pages are directly released.

In SBH execution, the size of hibernation file is an important factor affecting the resuming efficiency. To minimize the size of hibernation file, SBH swaps out all swappable pages to reduce the consumption of memory. It is because that as the consumption of memory is proportional to the size of hibernation file and decreasing the memory consumption reduces the size of hibernation file. After the system resumption, OS finds that some pages are not in the main memory because all the swappable pages are swapped out before the hibernation mode. When a program accesses these pages that are not in the main memory, OS reloads these memory pages from the flash memory into the main memory. In order to increase the I/O efficiency, when OS reads the page, it reads several blocks (/pages) nearby the said

block (/page) in the main memory (i.e., page cache). As the temporal locality and spatial locality the mechanism has a good hit ratio, it is a type of prefetch mechanism. Many studies have proposed effective methods of improving prefetch based on the file system behavior. This paper highlights the efficiency optimization of prefetch of swap space after SBH.

The swap-out behavior of SBH is different from the normal swap-out behavior of OS. The normal swap-out occurs when the memory is insufficient. According to the quantity of insufficient memory, OS generates the corresponding swap-out quantity. Thus, the quantity of pages written out is not large in each swap-out. The SBH writes out all swappable pages before the system enters the hibernation mode. Except for OS kernel, the other pages are almost all swappable pages, so the quantity of pages written out by SBH is very large. These pages can be reordered according to their characteristics written out by SBH, so as to maximize the efficiency of prefetch.



**Figure 1: cumulative statistics of total swap I/O and memory consumption**

“Temple Run”, a popular Android app, is used to demonstrate the possibility of improving efficiency of swap space by reordering requests of swap-outs. In Figure 1, the horizontal axis represents the time after booting, and the vertical axis represents the cumulative statistics of total swap I/O and memory consumption of Temple Run. The green line is the accumulated amount of used memory of Temple Run. At time point 0, the memory amount used by Temple Run is very small. It is because that the SBH writes out all swappable pages before the hibernation mode. After the resuming, Temple Run accesses memory and these accesses trigger swap-in. Let us take Android/Linux as an example. In each swap-in, Linux reads in 8 consecutive pages (/blocks). One swap-in I/O lets OS prefetch 7 blocks/pages to the page cache. The red line in the figure is the quantity of data that the system reads in (including all pages bring in by a swap-in). It can be seen that the system reads a large quantity of data within five seconds after booting. The finding showed that the only 50% (“Memory used” divided by “Total Swap I/O”) of total I/O (including fetching the said block/page and prefetching) contributes the performance of swap space. This study proposes a method to improve the hit ratio of prefetch, so as to shorten the response time of each application after booting.

### 3. RELATED WORK

Many fast booting methods [11-13] have been proposed, which are able to accelerate the booting time by reordering the execution flow, lazy loading system modules, and removing some unnecessary booting steps [13]. Chrome OS [11] removes many repeated booting steps of the system to speed up booting. The hibernation/resuming technology [6, 12, 14, 15] is also used to accelerate booting. These methods can be used to skip the complicated system initialization process by restoring the system to the last system state.

Lo et al.[6] proposed two features, “system working memory is much smaller than the system's total memory [16]”, and “random access speed of flash memories is close to sequential access speed [17]”, which can be used to develop a fast booting/resuming methods. Extremely small hibernation file is retrieved through sequential access of flash memory to accelerate booting time. However, the disadvantage is that the OS issues a large quantity of swap-in requests to load the working set of a running program. These swap-in requests would prolong the response time of a program.

Microsoft developed fast booting for Windows 8 [18], which is similar to the fast booting proposed by Lo et al.[6]. Before the system enters the hibernation mode, the memory used by the users is abandoned, and only the memory of kernel and system services is stored in the hibernation file. After booting, the login screen directly appears, and users have to restart the software programs. As compared to fast booting method proposed by Lo et al.[6], the main disadvantage is that the method cannot shorten the starting time of software programs.

Microsoft also developed ReadyDrive and ReadyBoost [19] for Windows Vista. The technology is based on the assumption that random access speed of flash memory is faster than that of the hard drive. ReadyDrive stores all data needed for booting in flash memory. During booting, the system uses flash memory to read needed data quickly to accelerate booting. On the other hand, ReadyBoost uses flash memory as a hard disk cache, in order to accelerate the loading of application programs after booting.

Literatures [20-22] suggested that flash memory has become popular in the recent years, which access characteristics differ from those of the traditional hard disks. If the system uses flash memory as the secondary storage, especially as a swap space, the access policy and mechanisms should be adjusted so as to meet the characteristics of flash memory. That finding contributed to improving the swap system, and the proposed method can effectively reduce the frequency of erasing the flash memory.

### 4. SYSTEM ARCHITECTURE AND METHODS

The proposed swap-space algorithm must satisfy the following objectives. The first objective is to optimize the loading time of each process after resuming. This is because the OS does not know the program that the user will execute after the computer has resumed. The second objective is to arrange these pages according to the time that a program may access them. Besides predicting whether the pages would be used again, they should be arranged according to the sequence of use. Thus, the time difference between the time point when the page/block is prefetched (time a) and the time point when the page/block is used

(time b) can be shortened (i.e. shortening the time difference between time points a and b). The prefetched page is prevented from being discarded from the cached page for being in page cache too long. Therefore, arranging these pages according to the time correlation can increase the hit ratio. The third objective is that the designed method must be efficient. The proposed method cannot sacrifice user experience (i.e., execution speed of a program) to improve the response time after resuming. The fourth objective is the limited use of the memory, because the algorithm is executed in OS kernel, unlike the user mode program that has huge virtual memory.

Based on the above conditions, the first step of our algorithm is described below. After resume, whenever the program has a page fault, the logical block address (LBA) of a block that the OS reads in is recorded in an ordered list of each program. This ordered list is called page fault list (PFL). The PFL records the information of page/block that each process swap-in, as well as the sequence of swap-in. Notice that the OS kernel would minimize the number of page fault and our algorithm is executed only if there is a page fault, so the influence on user mode program is very slight. PFL of each process can be recorded in the file system, so the use of kernel memory can be minimized.

The second step is executed when the system enters the hibernation mode. The SBH writes out all the swappable pages in the system via block I/O subsystem to the flash memory. The block I/O subsystem is modified, so that the swappable pages are not written in the swap space directly. These pages to be written out by the block I/O subsystem are kept in a waiting queue temporarily. When the SBH moves all swappable pages into the waiting queue, all the swappable pages in this queue are first sequenced according to the process id of each page, then according to the order of the page in the PFL. Finally, the sequenced swappable pages are written out to the swap space. Therefore, the pages of each program are ordered according to the read-in sequence after the latest resume of each program.

After the system resumes, if a page fault occurs and the page/block is read in from the flash memory, the OS reads in several consecutive pages at one time. These additionally read-in pages are stored in the page cache temporarily. This is a prefetch mechanism of OS kernel for swap space. Based on our algorithm, these consecutively read-in pages have high correlation. If these programs have similar memory access behaviors after each resume, this algorithm can increase the hit ratio of prefetch effectively.

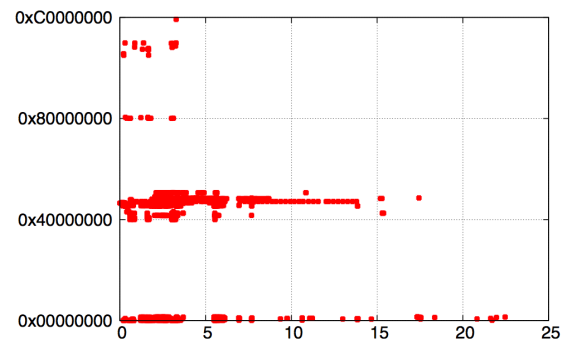


Figure 2.a: memory accessed of Temple Run

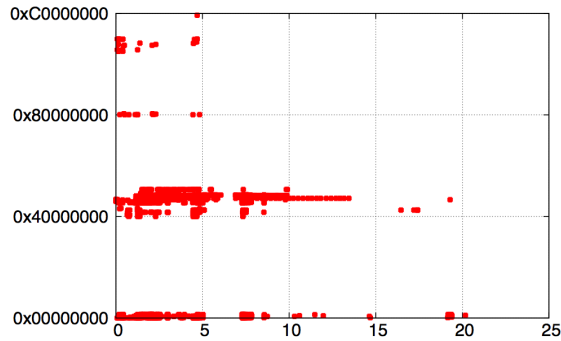


Figure 2.b: memory accessed of Temple Run

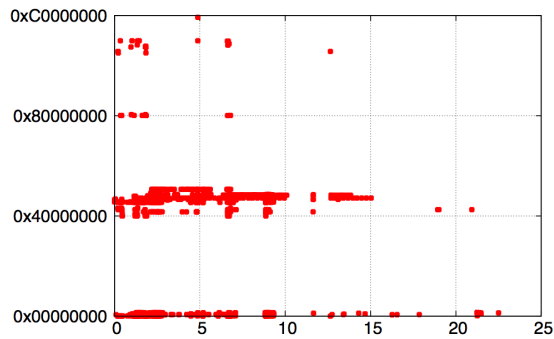


Figure 2.c: memory accessed of Temple Run

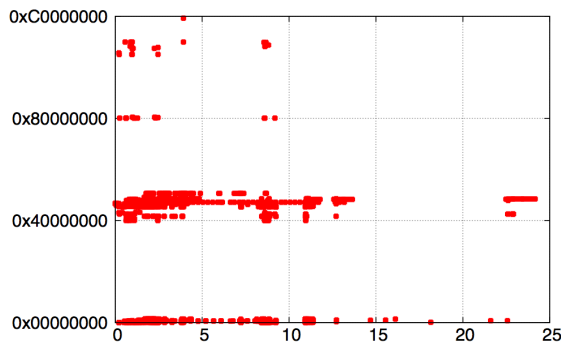


Figure 2.d: memory accessed of Temple Run

Figure 2.a-2.d shows the memory access behavior of the "Temple Run" app. The horizontal axis represents the time coordinate, zero represents the resume completion time, and the vertical axis represents the memory address. As shown in Figure 2, the memory access behaviors are very similar in the execution process after four times of resume of Temple Run. This proves that our algorithm can increase the performance effectively. The improvement of performance is discussed in the next section.

## 5. EXPERIMENTAL RESULTS

Android was used as operating environment for the experiment. Linux 2.6.32 and Android 2.2 were used. The hardware platform was Beagleboard-XM, and the CPU was ARM Cortex A8 with 512MB memory. The read/write characteristics of the flash memory for the experiment are shown in Table 1. As seen, when the data volume read/written each time is large, the reading speed

is high. When the read/written data volume exceeds 64KB, the reading/writing speed approaches to fixed values (20.29MB/s and 12.8MB/s respectively).

Table 1: Transcend microSDHC 8G class 10 (MB/s)

	4K	32K	64K	128K	256K
Read	5.03	15.57	20.29s	20.29s	20.35
Write	2.69	5.44	12.8	12.34	12.54

The function `do_swap_page()` of Linux kernel was modified, and all swap-in and swap-out in the kernel were recorded. Three applications were executed in the experiment, including Temple Run (Gaming), Gallery (performing a slide show) and Adobe PDF Reader (reading a PDF file). The three applications represent the three main application types in Android, which are game, image processing and office software. Three prefetch sizes were tested, including 8 pages (32KB), 16 pages (64KB) and 32 pages (128KB). The Linux preset prefetch size was 8 pages (32KB). However, in terms of the microSD for the experimental environment, 16 pages (64KB) were likely to perform the hardware performance.

The performance data measured included the hit ratio of page cache, the data volume read by OS kernel in the swap space, the number of major faults in the execution of applications, and the total I/O time. Figure 3 shows the four performance indexes. When the program fails to access memory, there is a page fault. The OS kernel checks whether the page has been in the page cache. If yes, it is a hit. The probability of hit is called hit ratio. If it is not in the page cache, Linux must read back this page in the flash memory. Such a page fault is called major fault. Therefore, the number of times that a major fault has occurred is the number of times that the Linux kernel has read the swap space. Figure 3 shows the major fault, and OS kernel must read back the dark gray data (the data that the said program accessed). Considering efficiency, Linux kernel prefetches the light gray data. The Linux kernel adopted in the experiment optimizes the flash memory, so the prefetched data are aligned with 32KB automatically. Linux puts the prefetched data in the page cache to increase the hit ratio. The I/O time is the time spent by Linux kernel on reading dark gray data and light gray data.

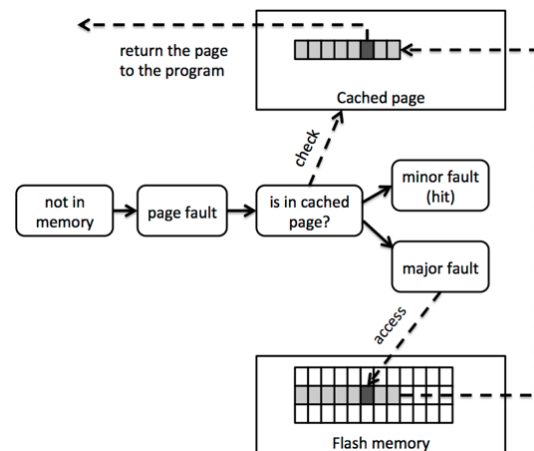


Figure 3: page fault handling of Linux

Figures 4~6 show the experimental results. The suspend/resume action was performed 10 times continuously for each experiment, so as to obtain more accurate data. The four methods were compared in each experiment. The “w/o ordering” used the preset swap space management style of Linux kernel, and the Linux preset prefetch size was 8 pages (32KB). “with ordering-8~32” uses our algorithm and different prefetch sizes (32KB~128KB). Take Temple Run as an example, the hit ratio increased from 60% to 90%, suggesting that the proposed method can avoid prefetching unnecessary data. The total swap I/O was improved by one third, and the improvement was mainly derived from an increased hit ratio. The total swap I/O increased slightly with the prefetch size. The quantity of major faults was improved considerably as the prefetch size increased. It is because that when the prefetch size increased from 32KB to 128KB, the hit ratio did not reduce significantly. The I/O time decreased from 3 seconds on average to 1.5 seconds, suggesting that the proposed method can reduce the response time of applications by half. The best performance of I/O time occurred in “with ordering-16 (64K)”. As shown in Table 1, it is because that the reading speeds of the microSD used for 64KB and 128KB are almost identical. Figure 5 and Figure 6 have similar results. The I/O time was improved by about 1/3.

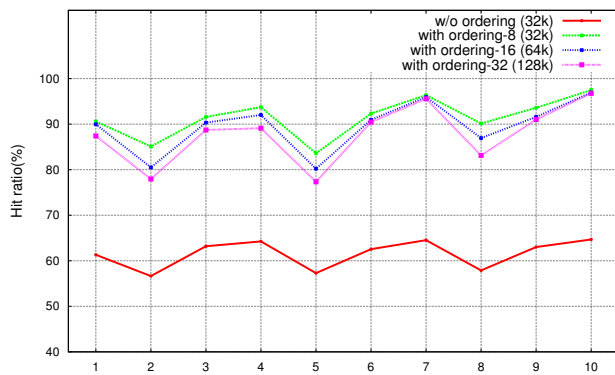


Figure 4.a: Temple run – Hit ratio

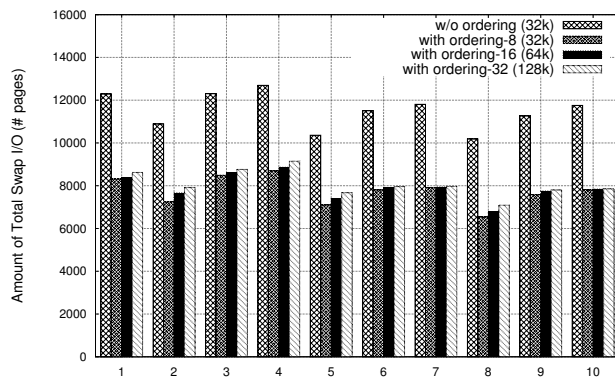


Figure 4.b: Temple run – Amount of Total Swap I/O

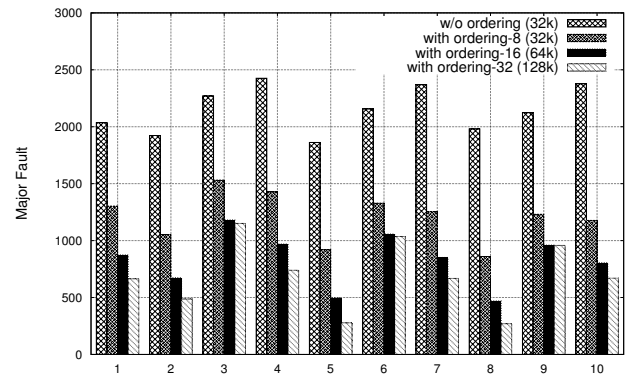


Figure 4.c: Temple run – Major Fault

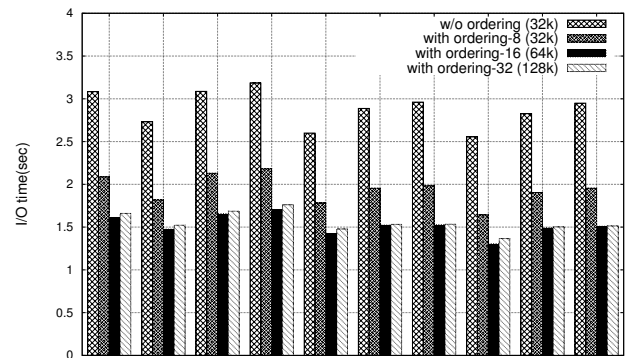


Figure 4.d: Temple run – I/O time

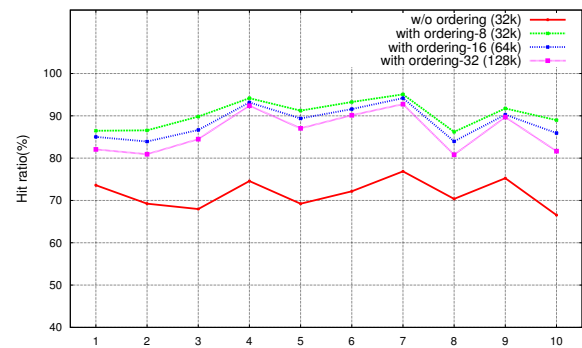


Figure 5.a: Gallery – Hit ratio

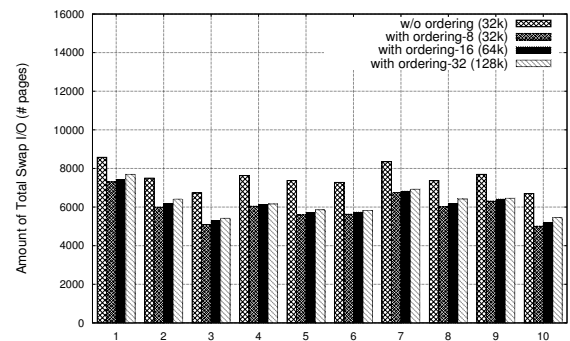


Figure 5.b: Gallery – Amount of Total Swap I/O

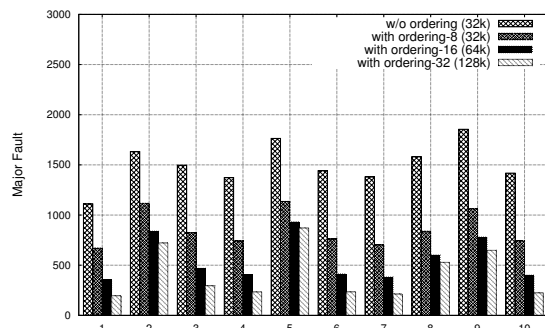


Figure 5.c: Gallery – Major fault

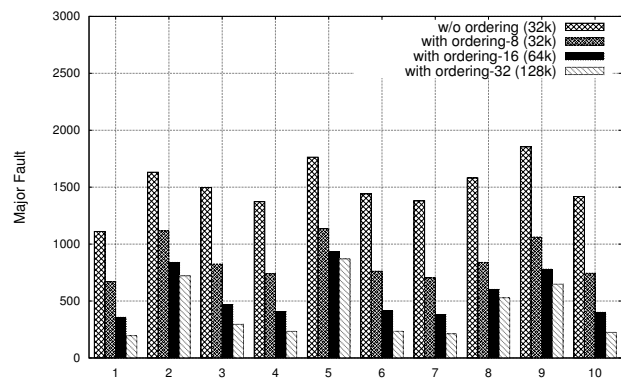


Figure 6.c: Adobe PDF Reader – Major fault

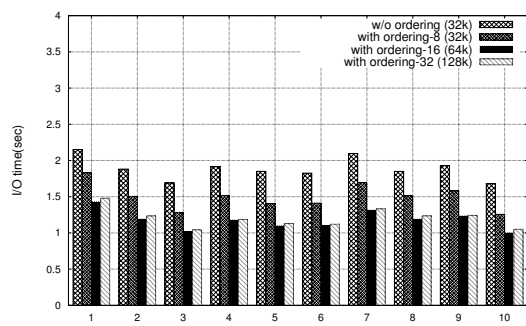


Figure 5.d: Gallery – I/O time

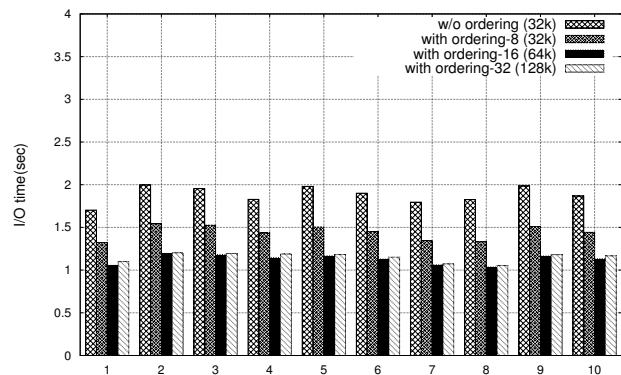


Figure 6.d: Adobe PDF Reader – I/O time

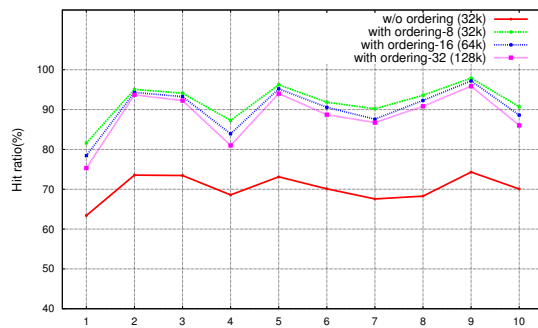


Figure 6.a: Adobe PDF Reader – Hit ratio

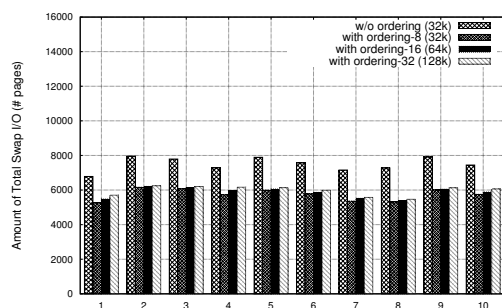


Figure 6.b: Adobe PDF Reader – Amount of Total I/O

## 6. CONCLUSIONS

Swap-before-Hibernate (SBH) can shorten the resume time of system, but the OS should swap in necessary data before continuing an application. This study found that the present swapping mechanism used by Linux is not suitable to SBH method. All the data to be written in the swap space were written out according to the sequence of use. The experimental results showed that the proposed method can reduce the I/O time by one third to one half. Therefore, the response time of applications after resuming was improved effectively.

## ACKNOWLEDGMENTS AND NAMES

This work was partially supported by the Ministry of Science and Technology, Taiwan, under project grant 103-2221-E-194 -019 -MY3. Authors' Chinese names are 羅習五、林泓逸、陳正元.

## 7. REFERENCES

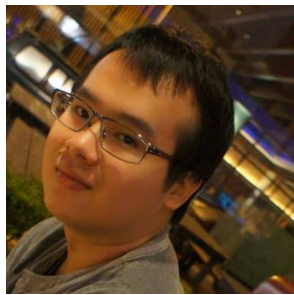
- [1] M. Wu and W. Zwaenepoel, "envy: A non-volatile, main memory storage system", in ASPLOS, 1994, pp. 86-97.
- [2] L. zhe Han, Y. Ryu, and K. S. Yim, "Cata: A garbage collection scheme for flash memory file system," in UIC, ser. Lecture Notes in Computer Science, vol. 4159. Springer, 2006, pp. 103-112.
- [3] G. Lawton, "Improved flash memory grows in popularity," IEEE Computer, vol. 39, no. 1, pp.16-18, 2006.

- [4] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?" *Behaviour & IT*, vol. 23, no. 3, pp. 153-163, 2004.
- [5] "Advanced configuration and power interface." [Online]. Available: [http://en.wikipedia.org/wiki/Advanced\\_Configuration\\_and\\_Power\\_Interface](http://en.wikipedia.org/wiki/Advanced_Configuration_and_Power_Interface)
- [6] S.-W. Lo, W.-S Tsai, J.-G Lin, and G.-S Cheng, "Swap-before-hibernate: a time efficient method to suspend an os to a flash drive," in *ACM SAC*, 2010, pp. 201-205.
- [7] V. Vellanki and A. Chervenak, "A cost-benefit scheme for high performance predictive prefetching," in *ACM/IEEE Supercomputing*, 1999, pp. 50.
- [8] J. Cooke, "Flash memory technology direction," Micron Applications Engineering Document, 2007.
- [9] Syuan-you Liao, "a data caching scheme for hybrid hard drives," thesis, pp. 1-53, 2011.
- [10] D. R. Hayes and S. Qureshi, "Microsoft vista: Serious challenges for digital investigations," *Proceedings of Student-Faculty Research Day*, pp. A3, 2008.
- [11] "Chromium os." [Online]. Available: <http://sites.google.com/a/chromium.org/dev/chromium-os/>
- [12] H. Kaminaga, "Improving linux startup time using software resume (and other techniques)," vol. 2, pp. 17.
- [13] C. Park, K. Kim, Y. Jang, and K. Hyun, "Linux bootup time reduction for digital still camera," vol. 2, pp. 231.
- [14] K. Baik, S. Kim, S. Woo, and J. Choi, "Boosting up embedded linux device: experience on linux-based smartphone," in *Proceedings of the Linux Symposium*, 2010, pp. 9-18.
- [15] C.-J. Chang, C.-W. Chang, C.-Y. Yang, Y.-H. Chang, C.-C. Pan, and T.-W. Kuo, "A run-time page selection methodology for efficient quality-based resuming," in *IEEE RTCSA*, pp.351-359.
- [16] P.J. Denning, "The working set model set model for program behavior," *Commun. ACM*, vol. 11, no.5, pp.323-333, 1968.
- [17] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for ssd performance," in *USENIX Annual Technical Conference*, 2008, pp. 57-70.
- [18] "Delivering fast boot times in windows 8." [On-line]. Available: <http://blogs.msdn.com/b/b8/archive/2011/09/08/delivering-fast-boot-times-in-windows-8.aspx>
- [19] "Windows vista i/o technologies." [Online]. Available: [http://en.wikipedia.org/wiki/Windows\\_Vista\\_I/O\\_technologies](http://en.wikipedia.org/wiki/Windows_Vista_I/O_technologies)
- [20] O. Kwon and K.Koh, "Swap-aware garbage collection for nand flash memory based embedded systems," in *IEEE CIT 2007*, pp.787-792.
- [21] S. Ko, S. Jun, Y. Ryu, O.Kwon, and K.Koh, "A new Linux swap system for flash memory storage devices," in *IEEE ICCSA 2008*, pp.151-156.
- [22] D.Jung, J.-s.Kim, S.-y.Park, J.-u.Kang, and J.Lee, "Fass: A flash-aware swap system," in *IWSSPS*, 2005.
- [23] "Standby power." [Online]. Available: <http://standby.lbl.gov/faq.html#muc>

## ABOUT THE AUTHORS:



Shiwu Lo received his B.S.E. degree in Computer Science and Information Engineering from Yuan Ze University in 1998. He received his M.S. degree in Computer Science and Information Engineering from National Chung-Cheng University and received Ph.D. degrees in Computer Science and Information Engineering from the National Taiwan University in 2000 and 2005, respectively.



Hung-Yi Lin received his M.S. degree in Computer Science and Information Engineering from National Chung-Cheng University.



Zhengyuan Chen received his M.S. degree in Computer Science and Information Engineering from National Chung-Cheng University.