Proceedings of the Linux Symposium

July 14–16th, 2014
Ottawa, Ontario
Canada

Contents

Btr-Diff: An Innovative Approach to Differentiate BtrFs Snapshots **7**
N. Mandliwala, S. Pimpale, N. P. Singh, G. Phatangare

Leveraging MPST in Linux with Application Guidance to Achieve Power and Performance Goals **13**
M.R. Jantz, K.A. Doshi, P.A. Kulkarni, H. Yun

CPU Time Jitter Based Non-Physical True Random Number Generator **23**
S. Müller

Policy-extendable LMK filter framework for embedded system **49**
K. Baik, J. Kim, D. Kim

Scalable Tools for Non-Intrusive Performance Debugging of Parallel Linux Workloads **63**
R. Schöne, J. Schuchart, T. Ilsche, D. Hackenberg

Veloces: An Efficient I/O Scheduler for Solid State Devices **77**
V.R. Damle, A.N. Palnitkar, S.D. Rangnekar, O.D. Pawar, S.A. Pimpale, N.O. Mandliwala

Dmdedup: Device Mapper Target for Data Deduplication **83**
V. Tarasov, D. Jain, G. Kuenning, S. Mandal, K. Palanisami, P. Shilane, S. Trehan, E. Zadok

SkyPat: C++ Performance Analysis and Testing Framework **97**
P.H. Chang, K.H. Kuo, D.Y. Tsai, K. Chen, Luba W.L. Tang

Computationally Efficient Multiplexing of Events on Hardware Counters **101**
R.V. Lim, D. Carrillo-Cisneros, W. Alkowaileet, I.D. Scherson

The *maxwell(8)* random number generator **111**
S. Harris

Popcorn: a replicated-kernel OS based on Linux **123**
A. Barbalace, B. Ravindran, D. Katz

Btr-Diff: An Innovative Approach to Differentiate BtrFs Snapshots

Nafisa Mandliwala
Pune Institute of Computer Technology
nafisa.mandliwala@gmail.com

Swapnil Pimpale
PICT LUG
pimpale.swapnil@gmail.com

Narendra Pal Singh
Pune Institute of Computer Technology
narendrapal2020@gmail.com

Ganesh Phatangare
PICT LUG
gphatangare@gmail.com

Abstract

Efficient storage and fast retrieval of data has always been of utmost importance. The BtrFs file system is a copy-on-write (COW) based B-tree file system that has an built-in support for snapshots and is considered a potential replacement for the EXT4 file system. It is designed specifically to address the need to scale, manage and administer large storage configurations of Linux systems. Snapshots are useful to have local online “copies” of the file system that can be referred back to, or to implement a form of deduplication, or for taking a full backup of the file system. The ability to compare these snapshots becomes crucial for system administrators as well as end users.

The existing snapshot management tools perform directory based comparison on block level in user space. This approach is generic and is not suitable for B-tree based file systems that are designed to cater to large storage. Simply traversing the directory structure is slow and only gets slower as the file system grows. With the BtrFs send/receive mechanism, the filesystem can be instructed to calculate the set of changes made between the two snapshots and serialize them to a file.

Our objective is to leverage the `send` part in the kernel to implement a new mechanism to list all the files that have been added, removed, changed or had their metadata changed in some way. The proposed solution takes advantage of the BtrFs B-tree structure and its powerful snapshot capabilities to speed up the tree traversal and detect changes in snapshots based on inode values. In addition, our tool can also detect changes between a snapshot and an explicitly mentioned parent. This lends itself for daily incremental backups of the file system, and can very easily be integrated with existing snapshot management tools.

1 Introduction

In current times, where data is critical to every organization, its appropriate storage and management is what brings in value. Our approach aims at taking advantage of the BtrFs architectural features that are made available by design. This facilitates speedy tree traversal to detect changes in snapshots. The `send` ioctl runs the tree comparison algorithm in kernel space using the on-disk metadata format (rather than the abstract `stat` format exported to the user space), which includes the ability to recognize when entire sub-trees can be skipped for comparison. Since the whole comparison algorithm runs in kernel space, the algorithm is clearly superior over existing user space snapshot management tools such as Snapper[1].

Snapper uses `diff` algorithm with a few more optimizations to avoid comparing files that have not changed. This approach requires all of the metadata for the two trees being compared to be read. The most I/O intensive part is not comparing the files but generating the list of changed files. It needs to list all the files in the tree and `stat` them to see if they have changed between the snapshots. The performance of such an algorithm degrades drastically as changes to the file system grow. This is mainly caused because Snapper deploys an algorithm that is not specifically designed to run on COW based file systems.

The rest of the paper is organized as follows: Section 2 explains the BtrFs internal architecture, associated data structures, and features. Working of the send-receive code and the `diff` commands used, are discussed in

[1]The description takes into consideration, the older version of Snapper. The newer version, however, does use the send-receive code for `diff` generation

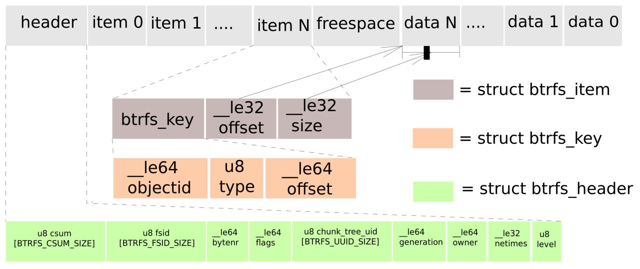


Figure 1: The Terminal Node Structure (src: BtrFs Design Wiki)

Section 3. Section 4 covers the design of the proposed solution, the algorithm used and its implementation. This is followed by benchmarking and its analysis in Section 5. Section 6 states applications of the send-receive code and `diff` generation. Section 7 lists the possible feature additions to Btr-diff. Finally, Section 8 summarizes the conclusions of the paper and is followed by references.

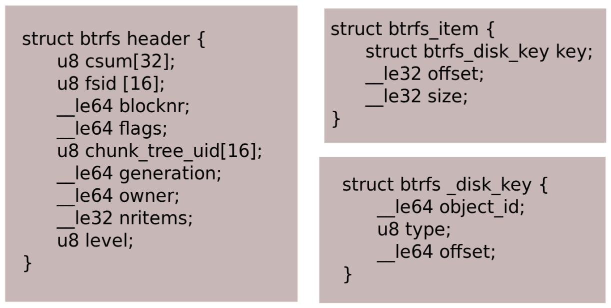
2 BtrFs File System

The Btrfs file system is scalable to a maximum file/file system size up to 16 exabytes. Although it exposes a plethora of features w.r.t. scalability, data integrity, data compression, SSD optimizations etc, this paper focuses only on the ones relevant to snapshot storage: comparison and retrieval.

The file system uses the 'copy-on-write' B-tree as its generic data structure. B-trees are used as they provide logarithmic time for common operations like insertion, deletion, sequential access and search. This COW-friendly B-tree in which the leaf node linkages are absent was originally proposed by Ohad Rodeh. In such trees, writes are never made in-place, instead, the modified data is written to a different location, and the corresponding metadata is updated. BtrFs, thus, has built-in support for snapshots, which are point-in-time copies of entire subvolumes, and rollbacks.

2.1 Data Structures

As seen in Figure 1, the BtrFs architecture consists of three data structures internally; namely `blockheader`, `key` and `item`. The `blockheader` contains checksums, file system specific `uuid`, the level at which the block is present in the tree etc. The `key`, which defines the order in the tree, has the fields: `objectid`, `type` and `offset`. Each subvolume has its own set of object ids. The type field contains the kind of item it is, of which, the prominent ones are `inode_item`, `inode_ref`, `xattr_item`, `orphan_item`, `dir_log_item`, `dir_item`, `dir_index`, `extent_data`, `root_item`, `root_ref`, `extent_item`, `extent_data_ref`, `dev_item`, `chunk_item` etc. The `offset` field is dependent on the kind of item.



```
struct btrfs header {
    u8 csum[32];
    u8 fsid [16];
    __le64 blocknr;
    __le64 flags;
    u8 chunk_tree_uid[16];
    __le64 generation;
    __le64 owner;
    __le32 nritems;
    u8 level;
}
```

```
struct btrfs_item {
    struct btrfs_disk_key key;
    __le32 offset;
    __le32 size;
}
```

```
struct btrfs _disk_key {
    __le64 object_id;
    u8 type;
    __le64 offset;
}
```

Figure 2: BtrFs Data Structures (src: BtrFs Design Wiki)

A typical leaf node consists of several items. `offset` and `size` tell us where to find the item in the leaf (relative to the start of the data area). The internal nodes contain `[key,blockpointer]` pairs whereas the leaf nodes contain a block header, array of fix sized items and the data field. Items and data grow towards each other. Typical data structures are shown in Figure 2.

2.2 Subvolumes and Snapshots

The BtrFs subvolume is a named B-tree that holds files and directories and is the smallest unit that can be snapshotted. A point-in-time copy of such a subvolume is called a snapshot. A reference count is associated with every subvolume which is incremented on snapshot creation. A snapshot stores the state of a subvolume and can be created almost instantly using the following command,

```
btrfs subvolume snapshot [-r] <source> [<dest>/]<name>
```

Such a snapshot occupies no disk space at creation. Snapshots can be used to backup subvolumes that can be used later for restore or recovery operations.

3 Send-Receive Code

Snapshots are primarily used for data backup and recovery. Considering the size of file system wide snapshots, detecting how the file system has changed between two given snapshots, manually, is a tedious task. Thus developing a clean mechanism to showcase differences between two snapshots becomes extremely important in snapshot management.

The `diff` command is used for differentiating any two files, but it uses text based[2] search which is time and computation intensive. Also, since it works on individual files, it does not give a list of files were modified/created/deleted on snapshot level. With the send/receive code, BtrFs can be instructed to calculate changes between the given snapshots and serialize them to a file. This file can later be replayed (on a BtrFs system) to regenerate one snapshot from another based on the instructions logged in the file.

As the name suggests, the send-receive code has a `send` side, that runs in kernel space and a `receive` side, which runs in user space. To calculate the difference between the two snapshots, the user simply gives a command line input given as follows:

```
btrfs send [-v] [-i <subvol>] [-p <parent>] <subvol>
```

`-v` : Enable verbose debug output. Each occurrence of this option increases the verbose level more.

`-i<subvol>` : Informs btrfs send that this subvolume, can be taken as 'clone source'. This can be used for incremental sends.

`-p<subvol>` : Disable automatic snaphot parent determination and use `<subvol>` as parent. This subvolume is also added to the list of 'clone sources'

`-f<outfile>` : Output is normally written to stdout. To write to a file, use this option. An alternative would be to use pipes.

Internally, this mechanism is implemented with the `BTRFS_IOC_SEND` ioctl() which compares two trees representing individual snapshots. This operation accepts a file descriptor representing a mounted volume and the subvolume ID corresponding to the snapshot of interest. It then calculates changes between the two given snapshots. The command sends the subvolume specified by `<subvol>` to `stdout`. By default, this will send the whole subvolume. The following are some more options for output generation:

- The operation can take a list of snapshot / subvolume IDs and generate a combined file for all of them. The parent snapshot can be specified explicitly. Thus, differences can be calculated with respect to a grandparent snapshot instead of a direct parent.
- The command also accepts 'clone sources' which are subvolumes that are expected to already exist on the receive side. Thus logging instructions for those subvolumes can be avoided and instead, only a 'clone' instruction can be sent. This reduces the size of the difference file.

The comparison works on the basis of meta-data. On detecting a metadata change, the respective trace is carried out and the corresponding instruction stream is generated. The output of the send code is an instruction stream consisting of `create/rename/link/write/clone/chmod/mkdir` instructions. For instance, consider that a new directory has been added to a file system whose snapshot has already been taken. If a new snapshot is then taken, the `send-receive` code will generate an instruction stream consisting of the instruction `mkdir`. The send receive code is thus more efficient as comparison is done only for changed files and not for the entire snapshots.

Taking into consideration the obvious advantages of using the send-receive code, the proposed solution uses it as a base for generating a `diff` between file system snapshots. To make the output of send code readable, we extract the data/stream sent from the send code and decode the stream of instructions into suitable format.

4 Design and Implementation

The proposed solution makes use of the BtrFs send ioctl `BTRFS_IOC_SEND` for diff generation. We have utilized the send-side of the send-receive code and extended the receive-side to give a view of the changes incurred to the file system between any two snapshots. This view

[2]`diff` does a line by line comparison of the given files, finds the groups of lines that vary, and reports each group of differing lines. It can report the differing lines in several formats, which serve different purposes.

```
Strategy: Go to the first items of both trees. Then do

If both trees are at level 0
  Compare keys of current items
    If left < right treat left item as new, advance left tree
      and repeat
    If left > right treat right item as deleted, advance right tree
      and repeat
    If left == right do deep compare of items, treat as changed if
      needed, advance both trees and repeat
If both trees are at the same level but not at level 0
  Compare keys of current nodes/leafs
    If left < right advance left tree and repeat
    If left > right advance right tree and repeat
    If left == right compare blockptrs of the next nodes/leafs
      If they match advance both trees but stay at the same level
        and repeat
      If they don't match advance both trees while allowing to go
        deeper and repeat
If tree levels are different
  Advance the tree that needs it and repeat

Advancing a tree means:
  If we are at level 0, try to go to the next slot. If
  that is not possible, go one level up and repeat. Stop when we found a level
  where we could go to the next slot. We may at this point be on a node or a
  leaf.

  If we are not at level 0 and not on shared tree blocks, go one level deeper.

  If we are not at level 0 and on shared tree blocks, go one slot to the right
  if possible or go up and right.
```

Figure 3: Tree Traversal Algorithm (src: BtrFs Send-Receive Code)

includes a list of the files and directories that underwent changes and also the file contents that changed between the two specified snapshots. Thus, a user would be able to view a file over a series of successive modifications.

The traversal algorithm is as given in Figure 3.

The BtrFs trees are ordered according to `btrfs_key` which contains the fields: `objectid`, `type` and `offset`. As seen in Figure 3, the comparison is based on this `key`. The 'right' tree is the old tree (before modification) and the 'left' tree is the new one (after modification). The comparison between left and right is actually a key comparison to check the ordering. When only one of the trees is advanced, the algorithm steps through to the next item and eventually one tree ends up at a later index than the other. The tree that reaches the end quicker, evidently, has missing entries which indicates a file deletion on this 'faster' side or a file creation on the 'slower'. The tree is advanced accordingly so that both the trees maintain almost the same level. The changes detected include changes in inodes, that is, addition and deletion of inodes, change in references to a file, change in extents and change in transaction ids that last touched a particular inode. Recording transaction IDs helps in maintaining file system consistency.

To represent the instruction stream in a clean way, we define the BtrFs `subvolume diff` command, as shown in Figure 4. This lists the files changed (added, modi-

```
btrfs subvolume diff [-p <snapshot1>] <snapshot2>

Output consists of:

I] A list of the files created, deleted
and modified (with corresponding number
of insertions and deletions).

II] Newly created files along with their
corresponding contents.

III] Files that have undergone write
modifications, with the corresponding
modified content in terms of blocks.
```

Figure 4: Btr-Diff and its output

fied, removed) between snapshot1 and snapshot2 where snapshot1 is optional. If snapshot1 is not specified, a `diff` between snapshot2 and its parent will be generated. The output generated by interpreting the data and executing the command above will be represented as given in Figure 4.

5 Performance and Optimization Achieved

Performance of Btr-diff was evaluated by varying the size of modifications done to a subvolume. Modifications of 1 GB to 4 GB were made and snapshots were taken at each stage. Btr-diff was then executed to generate a `diff` between these snapshots i.e. from 1–2 GB, 1–3 GB and so on. Benchmarking was done using the `time` command. The `time` command calculates the `real`, `sys` and `user` time of execution. The significance of each is as follows:

`real`: The elapsed real time between invocation and termination of the program. This is affected by the other processes and can be inconsistent.

`sys`: The system CPU time, i.e. the time taken by the program in the kernel mode.

`user`: The user CPU time, i.e. the time taken by the program in the user mode.

A combination of `sys` and `user` time is a good indicator of the tool's performance.

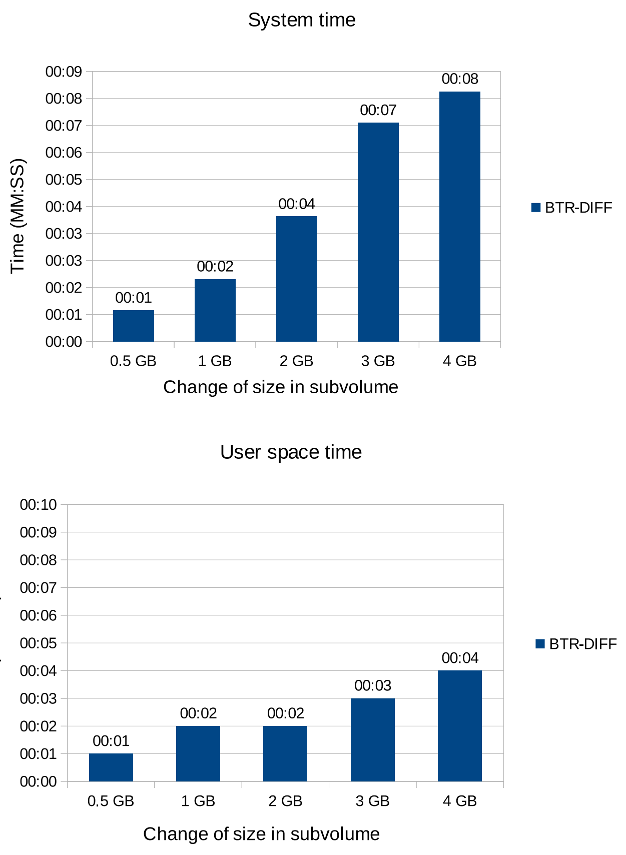


Figure 5: Time usage

The graphs in Figure 5 showcases the `sys` and `user` time performance of Btr-diff for different values of 'size differences' between the two given snapshots. The upper figure shows the time spent in kernel space by Btr-diff, while the lower figure shows the time spent in user space. The graphs clearly depict that the proposed method generates a `diff` in almost constant time. For changes of 1 GB, Btr-diff spends one second in the kernel and one second in user space. When these changes grow to 4 GB, Btr-diff maintains almost constant time and spends 8 seconds in the kernel and only 4 seconds in user space. Since the traversal is executed in kernel space, the switching between kernel and user space is reduced to a great extent. This has a direct implication on the performance.

There are various techniques/approaches that can be used for snapshot management. Using the send-receive based technique for BtrFs provides a lot of advantages over other generic comparison algorithms/techniques. Send/receive code is more efficient as its comparison is meant only for changed structure/files and not for entire snapshots. This reduces redundant comparisons.

6 Applications

Snapshot `diff` generation has several applications:

- Backups in various forms. Only the instruction stream can be stored and replayed at a later instant to regenerate subvolumes at the receiving side.
- File system monitoring. Changes made to the file system over a period of time can easily be viewed.
- A cron job could easily send a snapshot to a remote server on a regular basis, maintaining a mirror of a filesystem there.

7 Road Ahead

The possible optimizations/feature additions to Btr-diff could be as follows:

- Traversing snapshots and generating `diff` for a particular subvolume only.
- Btr-diff currently displays contents of modified files in terms of blocks. This output could be processed further and a `git-diff` like output can be generated which is more readable.

8 Conclusion

The lack of file system specific snapshot comparison tools for BtrFs have deterred the usage of its powerful snapshot capabilities to their full potential. By leveraging the in-kernel snapshot comparison algorithm (`send/receive`), a considerable reduction in the time taken for snapshot comparison is achieved. This is coupled with lower computation as well. The existing methods are generic and thus take longer than required to compute the diff. Our solution thus addresses this impending need for a tool that uses the BtrFs features to its advantage and gives optimum results.

References

[1] *Btrfs Main Page* http://btrfs.wiki.kernel.org/index.php/Main_Page

[2] Wikipedia - *Btrfs* http://en.wikipedia.org/wiki/Btrfs

[3] *Btrfs Design* https://btrfs.wiki.kernel.org/index.php/Btrfs_design#Snapshots_and_Subvolumes

[4] Linux Weekly News - *A short history of BtrFs* http://lwn.net/Articles/342892/

[5] IBM Research Report - *BTRFS:The Linux B-Tree Filesystem* http://domino.watson.ibm.com/library/CyberDig.nsf/1e4115aea78b6e7c85256b360066f0d4/6e1c5b6a1b6edd9885257a38006b6130!OpenDocument&Highlight=0,BTRFS

[6] Chris Mason - *Introduction to BtrFs* http://www.youtube.com/watch?v=ZW2E4WgPlzc

[7] Chris Mason - *BtrFs Filesystem: Status and New Features* http://video.linux.com/videos/btrfs-filesystem-status-and-new-features

[8] Avi Miller's BtrFs talk at LinuxConf AU http://www.youtube.com/watch?v=hxWuaozpe2I

[9] *Btrfs Data Structures* https://btrfs.wiki.kernel.org/index.php/Data_Structures

[10] Linux Weekly News - *Btrfs send/receive* http://lwn.net/Articles/506244/

[11] openSUSE - *Snapper* http://en.opensuse.org/Portal:Snapper

Leveraging MPST in Linux with Application Guidance to Achieve Power and Performance Goals

Michael R. Jantz
University of Kansas
mjantz@ittc.ku.edu

Kshitij A. Doshi
Intel Corporation
kshitij.a.doshi@intel.com

Prasad A. Kulkarni
University of Kansas
kulkarni@ittc.ku.edu

Heechul Yun
University of Kansas
heechul@ittc.ku.edu

Abstract

In this work, we describe an approach that improves collaboration between applications, the Linux kernel, and hardware memory subsystem (controllers and the DIMMs) in order to balance power and performance objectives, and we present details of its implementation using the Linux 2.6.32 kernel (x64) as base. The implementation employs ACPI memory power state table (MPST) to organize system memory into power domains according to rank information. An application programming interface (API) in our implementation allows applications to efficiently communicate various provisioning goals concerning groups of virtual ranges to the kernel. The kernel biases allocation and reclamation algorithms in line with the provisioning goals. The goals may vary over time; thus at one time, the applications may request high power efficiency; and at another time, they may ask for bandwidth or capacity reservations, and so on. This paper describes the framework, the changes for incorporating MPST information, policy modifications, and examples and use cases for invoking the new capabilities.

1 Introduction

Recent computing trends necessitate an increased focus on power and energy consumption and support for multi-tenant use cases. There is therefore a need to multiplex hardware efficiently and without performance interference. Advances in allocating CPU, storage and network resources have made it possible to meet competing service quality objectives while reducing power or energy demands[9, 8, 3]. In comparison to other resources, however, it is very challenging to obtain precise control over distribution of memory capacity, bandwidth, or power, when virtualizing and multiplexing system memory. Precisely controlling memory power and performance is difficult because these effects intimately depend upon the results of activities across multiple layers of the vertical execution stack, which are often not available at any single layer or component.

In an effort to simplify resource management within each layer, current systems often separate and abstract away information necessary to coordinate cross-layer activity. For example, the Linux kernel views physical memory as a single, large, contiguous array of physical addresses. The physical arrangement of memory modules, and that of the channels connecting them to processors, together with the power control domains are all opaque to the operating system's memory management routines. Without exposing this information to the upper-level software, it is very difficult to design schemes that coordinate application demands with the layout and architecture of the memory hardware.

The selection of physical pages to bind to application virtual addresses also has a significant impact on memory power and performance. Operating systems use heuristics that reclaim either the oldest, or the least recently touched, or least frequently used physical pages in order to fill demands. Over time, after repeated allocations and reclaims, there is no guarantee that a collection of intensely accessed physical pages would remain confined to a small number of memory modules (or DIMMs). Even if an application reduces its dynamic memory footprint, its memory accesses can remain spread out across sufficiently many memory ranks to keep any ranks from transitioning into a low-power state to save power. The layout and distribution of each application's hot pages not only affects the ability of memory modules to transition to lower power states during intervals of low activity, but also impacts the extent of interference caused by a program's activity in memory and the responsiveness experienced by other active programs. Thus a more discriminating approach than is available in current systems for multiplexing of physical memory is highly desirable.

Furthermore, data-intensive computing continues to raise demands on memory. Recent studies have shown that memory consumes up to 40% of total system power in enterprise servers [7] making memory power a dominant factor in overall power consumption. If an applications high-intensity accesses are concentrated among a small fraction of its total address space, then it is possible to achieve power-efficient performance by co-locating the active pages among a small fraction of DRAM banks. At the same time, an application that is very intensive in its memory accesses may prefer that pages in its virtual address span are distributed as widely as possible among independent memory channels to maximize performance. Thus, adaptive approaches are needed for improving power efficiency and performance isolation in the scheduling of memory.

We have designed and implemented a Linux kernel-based framework that improves collaboration between the applications, Linux kernel, and memory hardware in order to provide increased control over the distribution of memory capacity, bandwidth, and power. The approach employs the ACPI memory power state table (MPST)[1], which specifies the configuration's memory power domains and their associated physical addresses. Our modified Linux kernel leverages this information to organize physical memory pages into software structures (an abstraction called "trays") that capture the physically independent power domains. The modified kernel's page management routines perform all allocation and recycling over our software trays.

Our framework includes an application programming interface (API) that allows applications to efficiently communicate provisioning goals to the kernel by applying *colors* to portions of their virtual address space. A color is simply a hint applied to a virtual address range that indicates to the operating system that some common behavior or intention spans pages, even if the pages are not virtually contiguous. Applications can also associate *attributes* (or combinations of attributes) with each color. Attributes provide information (typically some intent or provisioning goal) to the operating system about how to manage the colored range. Colors and their associated attributes can be applied and changed at any time, and our modified Linux kernel attempts to interpret and take them into account while performing allocation, recycling, or page migration decisions.

We re-architect the memory management of a recent Linux kernel (x64, version 2.6.32) to implement our approach. Our recently published work, *A Framework for Application Guidance in Virtual Memory Systems* [6], describes the high-level design and intuition behind our approach and presents several experiments showing how it can be used to achieve various objectives, such as power savings and capacity provisioning. In this work, we provide further design and implementation details, including major kernel modifications and specific functions and tools provided by our coloring API.

The next section describes how our custom kernel leverages MPST information to construct trays and provides details of the structural and procedural modifications necessary to perform allocation and recycling over trays. In Section 3, we present our application programming interface to efficiently communicate application intents to our Linux kernel using colors, and we provide details of the kernel modifications that are necessary to receive and interpret this communication. In Section 4 we discuss our plans for future work, and Section 5 concludes the paper.

2 Leveraging MPST in the Linux Kernel

Our custom kernel organizes physical memory pages into the power-manageable tray abstraction by leveraging information provided by the ACPI memory power state table. Our implementation enables tray-based memory allocation and reclaim policies, which we describe in the following section.

2.1 Tray Design

Modern server systems employ a Non-Uniform Memory Access (NUMA) architecture which divides memory into separate regions (*nodes*) for each processor or set of processors. Within each NUMA node, memory is spatially organized into *channels*. Each channel employs its own memory controller and contains one or more DIMMs, which, in turn, each contain two or

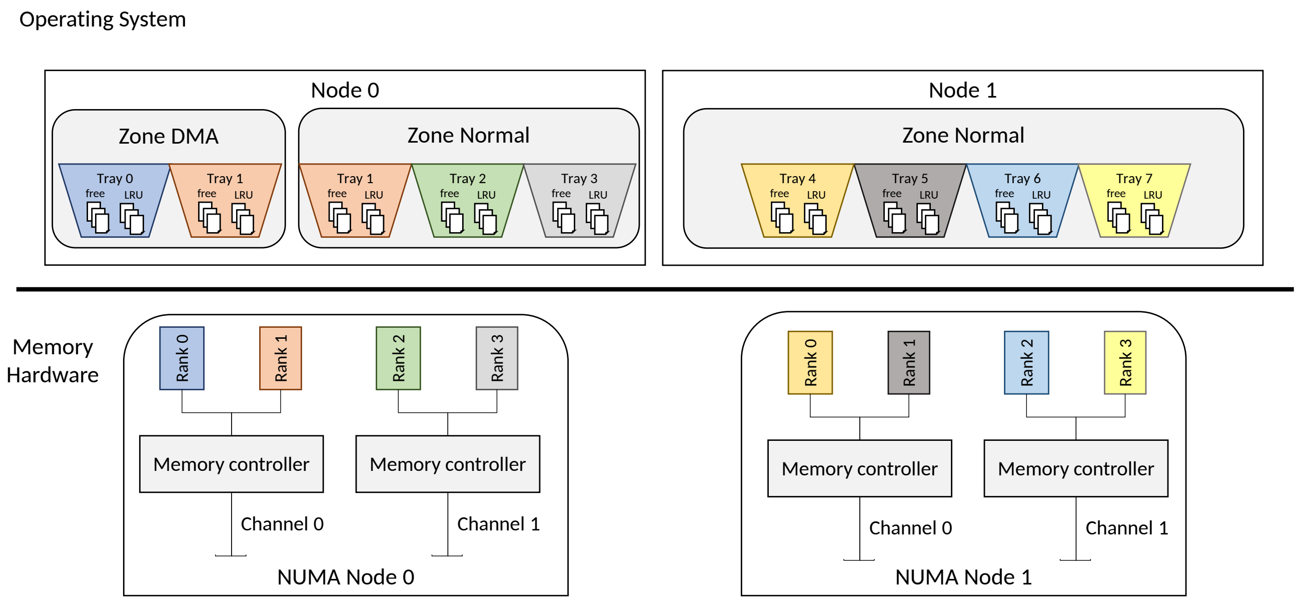


Figure 1: Organization of tray structures in relation to memory hardware

more *ranks*. Ranks comprise the actual memory storage and typically range from 2GB to 8GB in capacity. The memory hardware performs aggressive power management to transition from high power to low power states when either all or some portion of the memory is not active. Ranks are the smallest *power manageable unit*, which implies that transitioning between power states is performed at the rank level. Thus, different memory allocation strategies must consider an important power-performance tradeoff: distributing memory evenly across the ranks improves bandwidth which leads to better performance, while minimizing the number of active ranks consume less power.

The Linux kernel maintains a hierarchy of structures to represent and manage physical memory. *Nodes* in the Linux kernel correspond to the physical NUMA nodes in the hardware. Each node is divided into a number of blocks called *zones*, which represent distinct physical address ranges. At boot time, the OS creates physical page frames (or simply, *pages*) from the address range covered by each zone. Each page typically addresses 4KB of space. The kernel's physical memory management (allocation and recycling) operates on these pages, which are stored and tracked using various lists in each zone. For example, a set of lists of pages in each zone called the *free lists* describes all of the physical memory available for allocation.

To implement our approach, we create a new division in this hierarchy called *trays*. A tray is a software structure which contains sets of pages that reside on the same power-manageable memory unit. Each zone contains a set of trays and all the lists used to manage pages on the zone are replaced with corresponding lists in each tray. Figure 1 shows how our custom kernel organizes its representation of physical memory with trays in relation to the actual memory hardware.

One potential issue with the tray approach is that power-manageable domains that cross existing zone boundaries are represented as separate trays in different zones. Tray 1 in Figure 1 illustrates this situation. Another approach, proposed by A. Garg [5], introduces a *memory region* structure between the node and zone level to capture power-manageable domains. Although this approach is able to represent power-manageable domains that cross existing zone boundaries in a single structure, it requires a duplicate set of zone structures for each memory region. There are a number of important memory management operations that occur at the zone level and maintaining duplicate zone structures significantly complicates these operations. Our tray-based approach avoids zone duplication, and thus, avoids such complications. A more recent version of the memory region approach, proposed by S. Bhat [4], removes memory regions from the node-zone hierarchy entirely, and captures power-manageable domains in a data structure parallel to zones.

```
Flags (decoded below)      : 03
Node Enabled               : 1
Power Managed              : 1
Hot Plug Capable           : 0
Reserved                   : 00
Node ID                    : 0000
Length                     : 00000026
Range Address              : 0000000700000000
Range Length               : 000000013FFFFFFF
Num Power States           : 02
Num Physical Components    : 03
Reserved                   : 0000
...
```

Figure 2: Example MPST entry

2.2 Mapping Pages to Trays Using MPST

Assigning pages to the appropriate tray requires a mapping from physical addresses to the power-manageable units in hardware. We employ the ACPI defined memory power state table (MPST), which provides this mapping. Each entry in the MPST specifies a memory power node with its associated physical address ranges and supported memory power states. Some nodes may have multiple entries to support nodes mapped to non-contiguous address ranges.

The BIOS presents this information to the kernel at boot time in the form of an ACPI Data Table with statically packed binary data. Unfortunately, our kernel version does not include any facilities to parse the MPST into structured data. Therefore, we parse the table into text using utilities provided by the ACPI Component Architecture (ACPICA) project [2]. Figure 2 shows an example MPST entry as text. The most important fields in this table for defining power-manageable domains are `Range Address` and `Range Length`, which specify the physical address range of each memory power node. Thus, by either building this information into the kernel image or by copying it from user space during runtime, we are able to construct a global list of memory power nodes and their associated physical address ranges for use during memory management.

Pages can now be assigned to the appropriate tray by searching the global list of memory power nodes to find which node contains each page of memory. For most system configurations, this list is relatively short and searching it does not require significant overhead. Thus, to simplify our implementation effort, our custom kernel performs this search every time an operation needs to determine which tray should contain a particular page. An efficient implementation could perform this search once for each page and store information identifying the page's tray in the page flags field, similar to how the zone and node information for each page are currently stored.

It is important to note that our framework assumes that all of the physical addresses on each page correspond to memory on the same power-manageable unit in hardware. Some configurations interleave physical addresses within each page across power-manageable units in the hardware. For example, in an effort to exploit spatial locality, some systems interleave physical addresses across channels at the cache line boundary (typically 64 bytes). Supposing such a system has two DIMMs, each connected to its own channel, the first cache line within a page will use DIMM 0, the next will use DIMM 1, the next will use DIMM 0, and so on. In this case, our framework cannot control access to each individual DIMM, but may be able to control access to which ranks are used within the DIMMs.

2.3 Memory Management Over Trays

To enable memory management over trays, we modified our kernel's page management routines, which operate on lists of pages at the zone level, to operate over the same lists, but at a subsidiary level of trays. That is, zones are subdivided into trays, and page allocation, scanning, recycling are all performed at the tray level. For example, during page allocation, once a zone that is sufficient for a particular allocation has been found, the allocator calls `buffered_rmqueue` to find a page and remove it from the zone's free lists. In our custom kernel, `buffered_rmqueue` is modified to take an additional argument describing which tray's free lists should be searched, and each call site is modified to call `buffered_rmqueue` repeatedly, with each tray, until a suitable page is found. Most of the other required changes are similarly straightforward.

One notable complication has to do with the low-level page allocator in Linux, known as the buddy allocator. In order to quickly fulfill requests for contiguous ranges of pages, the Linux buddy allocator automatically groups contiguous blocks of pages. The *order* of a block of pages refers to the number of contiguous

```
struct tray {
  ...
  /*
   * free lists of pages of different orders
   */
  struct free_area    free_area[MAX_ORDER];
  ...
};
struct free_area {
  struct list_head    free_list[MIGRATE_TYPE];
  unsigned long       nr_free;
};
```

Figure 3: Definition of free lists in the buddy system

pages in that block, where an order-n block contains 2^n contiguous pages. Block orders range from 0 (individual pages) to some pre-defined maximum order (11 on our Linux 2.6.32 x64 system). Blocks of free pages are stored in a set of lists, where each list contains blocks of pages of the same order. In our implementation, the lists are defined at the tray-level as shown in Figure 3. When two contiguous order-n blocks of pages are both free, the buddy allocator removes these two blocks from the order-n free list and creates a new block to store on the order-$n+1$ free list. However, if two contiguous order-n blocks reside in different trays, their combined order-$n+1$ block cannot be placed on either tray (as it would contain pages belonging to the other tray). For this reason, we maintain a separate set of free lists, defined at the zone level, to hold higher order blocks that contain pages from separate trays. With this structure, our custom kernel is able to fulfill requests for low-order allocations from a particular power-manageable domain, while, at the same time, it is also able to handle requests for large blocks of contiguous pages, which may or may not reside on the same power-manageable unit.

3 Application Guidance under Linux

The goal of our framework is to provide control over memory resources such as power, bandwidth, and capacity in a way that allows the system to flexibly adapt to shifting power and performance objectives. By organizing physical memory into power-manageable domains, our kernel patch provides crucial infrastructure for enabling fine-grained resource management in software. However, memory power and performance depend not only on how physical pages are distributed across the memory hardware, but also on how the operating system binds virtual pages to physical pages, and on the demands and usage patterns of the upper-level applications. Thus, naïve attempts to manage these effects are likely to fail.

Our approach is to increase collaboration between the applications and operating system by allowing applications to communicate how they intend to use memory resources. The operating system interprets the application's intents and uses this information to guide memory management decisions. In this section, we describe our *memory coloring* interface that allows applications to communicate their intents to the OS, and the kernel modifications necessary to receive, interpret, and implement application intents.

3.1 Memory Coloring Overview

A color is an abstraction which allows the application to communicate to the OS hints about how it is going to use memory resources. Colors are sufficiently general as to allow the application to provide different types of performance or power related usage hints. In using colors, application software can be entirely agnostic about how virtual addresses map to physical addresses and how those physical addresses are distributed among memory modules. By coloring any N different virtual pages with the same color, an application communicates to the OS that those N virtual pages are alike in some significant respect, and by associating one or more attributes with that color, the application invites the OS to apply any discretion it may have in selecting the physical page frames for those N virtual pages.

By specifying coloring hints, an application provides a usage map to the OS, and the OS consults this usage map in selecting an appropriate physical memory scheduling strategy for those virtual pages. An application that uses no colors and therefore provides no guidance is treated normally – that is, the OS applies some default strategy. However, when an application provides guidance through coloring, depending on the particular version of the operating system, the machine configuration (such as how much memory and how finely interleaved it is), and other prevailing run time conditions in the machine, the OS may choose to veer a little or a lot from the default strategy.

System Call	Arguments	Description
`mcolor`	`addr`, `size`, `color`	Applies `color` to a virtual address range of length `size` starting at `addr`
`get_addr_mcolor`	`addr`,`*color`	Returns the current color of the virtual address `addr`
`set_task_mcolor`	`color`	Applies `color` to the entire address space of the calling process
`get_task_mcolor`	`*color`	Returns the current color of the calling process' address space
`set_mcolor_attr`	`color`, `*attr`	Associates the attribute `attr` with `color`
`get_mcolor_attr`	`color`, `*attr`	Returns the attribute currently associated with `color`

Table 1: System calls provided by the memory coloring API

3.2 Memory Coloring Example

The application interface for communicating colors and intents to the operating system uses a combination of configuration files, library software, and system calls. Let us illustrate the use of our memory coloring API with a simple example.

Suppose we have an application that has one or more address space extents in which memory references are expected to be relatively infrequent (or uniformly distributed, with low aggregate probability of reference). The application uses a color, say *blue* to color these extents. At the same time, suppose the application has a particular small collection of pages in which it hosts some frequently accessed data structures, and the application colors this collection *red*. The coloring intent is to allow the operating system to manage these sets of pages more efficiently – perhaps it can do so by co-locating the *blue* pages on separately power-managed units from those where *red* pages are located, or, co-locating *red* pages separately on their own power-managed units, or both. A possible second intent is to let the operating system page the *blue* ranges more aggressively, while allowing pages in the *red* ranges an extended residency time. By locating *blue* and *red* pages among a compact group of memory ranks, an operating system can increase the likelihood that memory ranks holding the *blue* pages can transition more quickly into self-refresh, and that the activity in *red* pages does not spill over into those ranks. Since many usage scenarios can be identified to the operating system, we define "intents" and specify them using configuration files. A configuration file for this example is shown in Figure 4. In this file, the intents labeled `MEM-INTENSITY` and `MEM-CAPACITY` can capture two intentions: (a) that red pages are hot and blue pages are cold, and (b) that about 5% of application's dynamic resident set size (RSS) should fall into red pages, while, even though there are many blue pages, their low probability of access is indicated by their 3% share of the RSS. Next, let us examine exactly how these colors and intents are actually communicated to the operating system using system calls.

```
# Specification for frequency of reference:
INTENT   MEM-INTENSITY

# Specification for containing total spread:
INTENT   MEM-CAPACITY

# Mapping to a set of colors:
MEM-INTENSITY RED   0 //hot pages
MEM-CAPACITY  RED   5 //hint - 5% of RSS

MEM-INTENSITY BLUE 1 //cold pages
MEM-CAPACITY  BLUE 3 //hint - 3% of RSS
```

Figure 4: Example config file for colors and intents

3.3 System Calls in the Memory Coloring API

The basic capabilities of actually applying colors to virtual address ranges and binding attributes to colors are provided to applications as system calls. Applications typically specify colors and intents using configuration files as shown in the example above, and then use a library routine to convert the colors and intents into system call arguments at runtime.

Table 1 lists and describes the set of system calls provided by our memory coloring API. To attach colors to virtual addresses, applications use either `mcolor` (to color a range of addresses), or `set_task_mcolor` (to color their entire address space). Colors are represented as unique integers, and can be overlapped. Thus, we use an integer bit field to indicate the set of colors that have been applied to each address range. To implement `mcolor`, we add a color field to the `vm_area_struct`

```
struct mcolor_attr {
  unsigned int  intent[MAX_INTENTS];
  unsigned int  mem_intensity;
  float         mem_capacity;
};
```

Figure 5: Example attribute structure definition

kernel structure. In the Linux kernel, each process' address space is populated by a number of regions with distinct properties. These regions are each described by an instance of `vm_area_struct`. When an application calls `mcolor`, the operating system updates the `vm_area_struct` that corresponds to the given address range (possibly splitting an existing `vm_area_struct` and/or creating a new `vm_area_struct`, if necessary) to indicate that the region is colored. The `set_task_mcolor` implementation is similar: the `task_struct` kernel structure is modified to include a color field, and `set_task_mcolor` updates this field. In the case that the `vm_area_struct` and `task_struct` color fields differ, the operating system may attempt to resolve any conflicts. In our default implementation, we simply choose the color field on the `vm_area_struct` if it has been set.

Colors are associated with attributes using the `set_mcolor_attr` system call. To represent attributes, we use a custom structure called `mcolor_attr`. For each color, this structure packages all of the data necessary to describe the color's associated intents. Figure 5 shows the `mcolor_attr` definition that is used for the example in Section 3.2. The `intent` field indicates whether a particular type of intent has been specified, and the remaining fields specify data associated with each intent. The kernel maintains a global table of colors and their associated attributes. When the application calls `set_mcolor_attr`, the kernel links the given attribute to the color's position in the global table. Note that in this implementation, there is only one attribute structure associated with each color. We bind multiple intents to one color by packaging the intents together into a single attribute structure.

3.4 Interpreting Colors and Intents During Memory Management

Lastly, we examine exactly how our modified Linux kernel steers its physical memory management decisions in accordance with memory coloring guidance. To describe this process, let us again consider the example in Section 3.2. Before the example application applies any colors, the system employs its default memory management strategy for all of the application's virtual pages. After applying the *red* and *blue* colors and binding these to their associated intents, the system will eventually fault on a colored page. Early during the page fault handling, the system determines the color of the faulting page (by examining the color field on the page's `vm_area_struct`) and looks up the color's associated attribute structure in the global attribute table. Now, the OS can use the color and attribute information to guide its strategy when selecting which physical page to choose to satisfy the fault.

For example, in order to prevent proliferation of frequently accessed pages across many power-manageable units, the operating system might designate one or a small set of the power-manageable domains (i.e. tray software structures) as the only domains that may be used to back *red* pages. Then, when the system faults on a page colored *red*, the OS will only consider physical pages from the designated set of domains to satisfy the fault. As another example, let us consider how the system might handle the `MEM-CAPACITY` intent. When the OS determines that pages of a certain color make up more than some percentage of the application's current RSS, then the system could choose to recycle frames containing pages of that color in order to fill demands. In this way, the system is able to fill demands without increasing the percentage of colored pages in the resident set.

Note that the specializations that an OS may support need not be confined just to selection of physical pages to be allocated or removed from the application's resident set. The API is general enough to allow other options such as whether or not to fault-ahead or to perform read-aheads or flushing writes, or whether or not to undertake migration of active pages from one set of memory banks to another in order to squeeze the active footprint into fewest physical memory modules. In this way, an OS can achieve performance, power, I/O, or capacity efficiencies based on guidance that application tier furnishes through coloring.

4 Future Work

While our custom kernel and API enable systems to design and achieve flexible, application-guided, power-aware management of memory, we do not yet have an understanding of what sets of application guidance and memory power management strategies will be most useful for existing workloads. Furthermore, our current API requires that all coloring hints be manually inserted into source code and does not provide any way to *automatically* apply beneficial application guidance. Thus, we plan to develop a set of tools to profile, analyze, and automatically control memory usage for applications. Some of the capabilities we are exploring include: (a) a set of tools and library software for applications to query detailed memory usage statistics for colored regions, (b) on-line techniques that adapt memory usage guidance based on feedback from the OS, and (c) integration with compiler and runtime systems to automatically partition and color the application's address space based on profiles of memory usage activity.

Simultaneously, we plan to implement color awareness in selected open source database, web server, and J2EE software packages, so that we can exercise complex, multi-tier workloads at the realistic scale of server systems with memory outlays reaching into hundreds of gigabytes. In these systems, memory power can reach nearly half of the total machine power draw, and therefore they provide an opportunity to explore dramatic power and energy savings from application-engaged containment of memory activities. We also plan to explore memory management algorithms that maximize performance by biasing placement of high value data so that pages in which performance critical data resides are distributed widely across memory channels. We envision that as we bring our studies to large scale software such as a complex database, we will inevitably find new usage cases in which applications can guide the operating system with greater nuance about how certain pages should be treated differently from others.

5 Conclusions

There is an urgent need for computing systems that are able to multiplex memory resources efficiently while also balancing power and performance tradeoffs. We have presented the design and implementation of a Linux-based approach that improves collaboration between the application, operating system, and hardware layers in order to provide a fine-grained, flexible, power-aware provisioning of memory. The implementation leverages the ACPI memory power state table to organize the operating system's physical memory pages into power-manageable domains. Additionally, our framework provides an API that enables applications to express a wide range of provisioning goals concerning groups of virtual ranges to the kernel. We have described the kernel modifications to organize and manage physical memory in software structures that correspond to power-manageable units in hardware. Finally, we have provided a detailed description of our API for communicating provisioning goals to the OS, and we have presented multiple use cases of our approach in the context of a realistic example.

References

[1] Advanced configuration and power interface specification, 2011. http://www.acpi.info/spec.htm.

[2] Acpi component architecture (acpica), 2013. http://www.acpi.info/spec.htm.

[3] Vlasia Anagnostopoulou, Martin Dimitrov, and Kshitij A. Doshi. Sla-guided energy savings for enterprise servers. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 120–121, 2012.

[4] Srivatsa S. Bhat. mm: Memory power management, 2013. http://lwn.net/Articles/546696/.

[5] Ankita Garg. mm: Linux vm infrastructure to support memory power management, 2011. http://lwn.net/Articles/445045/.

[6] Michael R. Jantz, Carl Strickland, Karthik Kumar, Martin Dimitrov, and Kshitij A. Doshi. A framework for application guidance in virtual memory systems. In *VEE '13: Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, March 2013.

[7] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12):39–48, December 2003.

[8] Lanyue Lu, P.J. Varman, and K. Doshi. Decomposing workload bursts for efficient storage resource management. *IEEE Transactions on Parallel and Distributed Systems*, 22(5):860 –873, may 2011.

[9] H. Wang, K. Doshi, and P. Varman. Nested qos: Adaptive burst decomposition for slo guarantees in virtualized servers. *Intel Technology Journal*, June, 16:2 2012.

CPU Time Jitter Based Non-Physical True Random Number Generator

Stephan Müller
atsec information security
`Stephan.Mueller@atsec.com`

Abstract

Today's operating systems provide non-physical true random number generators which are based on hardware events. With the advent of virtualization and the ever growing need of more high-quality entropy, these random number generators reach their limits. Additional sources of entropy must be opened up. This document introduces an entropy source based on CPU execution time jitter. The design and implementation of a non-physical true random number generator, the CPU Jitter random number generator, its statistical properties and the maintenance and behavior of entropy is discussed in this document.

The complete version of the analysis together with large amounts of test results is provided at www.chronox.de.

1 Introduction

Each modern general purpose operating system offers a non-physical true random number generator. In Unix derivatives, the device file `/dev/random` allows user space applications to access such a random number generator. Most of these random number generators obtain their entropy from time variances of hardware events, such as block device accesses, interrupts triggered by devices, operations on human interface devices (HID) like keyboards and mice, and other devices.

Limitations of these entropy sources are visible. These include:

- Hardware events do not occur fast enough.
- Virtualized environments remove an operating system from direct hardware access.
- Depending on the usage environment of the operating system, entire classes of hardware devices may be missing and can therefore not be used as entropy source.
- The more and more often used Solid State Disks (SSDs) advertise themselves as block devices to the operating system but yet lack the physical phenomenon that is expected to deliver entropy.
- On Linux, the majority of the entropy for the `input_pool` behind `/dev/random` is gathered from the `get_cycles` time stamp. However, that time stamp function returns 0 hard coded on several architectures, such as MIPS. Thus, there is not much entropy that is present in the entropy pool behind `/dev/random` or `/dev/urandom`.
- Current cache-based attacks allow unprivileged applications to observe the operation of other processes, privileged code as well as the kernel. Thus, it is desirable to have fast moving keys. This applies also to the seed keys used for deterministic random number generators.

How can these challenges be met? A new source of entropy must be developed that is not affected by the mentioned problems.

This document introduces a non-physical true random number generator, called CPU Jitter random number generator, which is developed to meet the following goals:

1. The random number generator shall only operate on demand. Other random number generators constantly operate in its lifetime, regardless whether the operation is needed or not, binding computing resources.
2. The random number generator shall always return entropy with a speed that satisfies today's requirement for entropy. The random number generator shall be able to be used synchronously with the entropy consuming application, such as the seeding of a deterministic random number generator.

3. The random number generator shall not block the request for user noticeable time spans.

4. The random number generator shall deliver high-quality entropy when used in virtualized environments.

5. The random number generator shall not require a seeding with data from previous instances of the random number generator.

6. The random number generator shall work equally well in kernel space and user space.

7. The random number generator implementation shall be small, and easily understood.

8. The random number generator shall provide a decentralized source of entropy. Every user that needs entropy executes its own instance of the CPU Jitter random number generator. Any denial of service attacks or other attacks against a central entropy source with the goal to decrease the level of entropy maintained by the central entropy source is eliminated. The goal is that there is no need of a central `/dev/random` or `/dev/urandom` device.

9. The random number generator shall provide perfect forward and backward secrecy, even when the internal state becomes known.

Apart from these implementation goals, the random number generator must comply with the general quality requirements placed on any (non-)physical true random number generator:

Entropy The random numbers delivered by the generator must contain true information theoretical entropy. The information theoretical entropy is based on the definition given by Shannon.

Statistical Properties The random number bit stream generated by the generator must not follow any statistical significant patterns. The output of the proposed random number generator must pass all standard statistical tools analyzing the quality of a random data stream.

These two basic principles will be the guiding central theme in assessing the quality of the presented CPU Jitter random number generator.

The document contains the following parts:

- Discussion of the noise source in Section 2
- Presentation of CPU Jitter random number generator design in Section 3
- Discussion of the statistical properties of the random number generator output in Section 4
- Assessment of the entropy behavior in the random number generator in Section 5

But now away with the theoretical blabber: show me the facts! What is the central source of entropy that is the basis for the presented random number generator?

2 CPU Execution Time Jitter

We do have deterministically operating CPUs, right? Our operating systems behave fully deterministically, right? If that would not be the case, how could we ever have operating systems using CPUs that deliver a deterministic functionality.

Current hardware supports the efficient execution of the operating system by providing hardware facilities, including:

- CPU instruction pipelines. Their fill level have an impact on the execution time of one instruction. These pipelines therefore add to the CPU execution timing jitter.
- The timer tick and its processing which alters the caches.
- Cache coherency strategies of the CPU with its cores add variances to instruction execution time as the cache controlling logic must check other caches for their information before an instruction or memory access is fetched from the local cache.
- The CPU clock cycle is different than the memory bus clock speed. Therefore, the CPU has to enter wait states for the synchronization of any memory access where the time delay added for the wait states adds to time variances.
- The CPU frequency scaling which alters the processing speed of instructions.
- The CPU power management which may disable CPU features that have an impact on the execution speed of sets of instructions.

In addition to the hardware nondeterminism, the following operating system caused system usage adds to the non-deterministic execution time of sets of instructions:

- Instruction and data caches with their varying information – tests showed that before the caches are filled with the test code and the CPU Jitter random number generator code, the time deltas are bigger by a factor of two to three;
- CPU topology and caches used jointly by multiple CPUs;
- CPU frequency scaling depending on the work load;
- Branch prediction units;
- TLB caches;
- Moving of the execution of processes from one CPU to another by the scheduler;
- Hardware interrupts that are required to be handled by the operating system immediately after the delivery by the CPU regardless what the operating system was doing in the mean time;
- Large memory segments whose access times may vary due to the physical distance from the CPU.

2.1 Assumptions

The CPU Jitter random number generator is based on a number of assumptions. Only when these assumptions are upheld, the data generated can be believed to contain the requested entropy. The following assumptions apply:

- Attacker having hardware level privileges are assumed to be not present. With hardware level privilege, on some CPU it may be possible to change the state of the CPU such as that caches are disabled. In addition, millicode may be changed such that operations of the CPU are altered such that operations are not executed any more. The assumption is considered to be unproblematic, because if an attacker has hardware level privilege, the collection of entropy is the least of our worries as the attacker may simply bypass the entropy collection and furnish a preset key to the entropy-seeking application.
- Attacker with physical access to the CPU interior is assumed to be not present. In some CPUs, physical access may allow enabling debug states or the readout of the entire CPU state at one particular time. With the CPU state, it may be possible to deduct upcoming variations when the CPU Jitter random number generator is executed immediately after taking a CPU state snapshot. An attacker with this capability, however, is also able to read out the entire memory. Therefore, when launching the attack shortly after the entropy is collected, the attacker could read out the key or seed material, bypassing the the entropy collection. Again, with such an attacker, the entropy collection is the least of our worries in this case.
- The CPU Jitter random number generator is always executed on CPUs connected to peripherals. When the CPU has no peripherals, including no access to RAM or any busses, special software can be expected to execute on the CPU fully deterministically. However, as this scenario requires a highly specialized environment that does not allow general purpose computing, this scenario is not applicable.

2.2 Jitter Depicted

With the high complexity of modern operating systems and their big monolithic kernels, all the mentioned hardware components are extensively used. However, due to the complexity, nobody is able to determine which is the fill level of the caches or branch prediction units, or the precise location of data in memory at one given time.

This implies that the execution of instruction may have miniscule variations in execution time. In addition, modern CPUs have a high-resolution timer or instruction counter that is so precise that they are impacted by these tiny variations. For example, modern x86 CPUs have a TSC clock whose resolution is in the nanosecond range.

These variations in the execution time of an identical set of CPU instructions can be visualized. For the sample code sequence given in Figure 1, the variation in time is shown in Figure 2.

The contents of the variable `delta` is not identical between the individual loop iterations. When running the

```
static inline void jent_get_nstime(uint64_t *out)
{
...
        if (clock_gettime(CLOCK_REALTIME, &time) == 0)
...
}

void main(void)
{
...
        jent_get_nstime(&time);
        jent_get_nstime(&time2);
        delta = time2 - time;
...
}
```

Figure 1: Sample code for time variance observation

code with a loop count of 1,000,000 on an otherwise quiet system to avoid additional time variance from the noise of other processes, we get data as illustrated in Figure 2.

Please note that the actual results of the aforementioned code contains a few exceptionally large deltas as an operating system can never be fully quiesced. Thus, the test results were processed to cut off all time deltas above 64. The limitation of the graph to all variations up to 64 can be considered as a "magnification" of the data set to the interesting values.

Figure 2 contains the following information of interest to us:

- The bar diagram shows the relative frequency of the different delta values measured by the code. For example, the delta value of 22 (nanoseconds – note the used timer returns data with nanosecond precision) was measured at 25% of all deltas. The value 23 (nanoseconds) was measured at about 25% of all time deltas.
- The red and blue vertical lines indicate the mean and median values. The mean and median is printed in the legend below the diagram. Note, they may overlap each other if they are too close. Use the legend beneath the diagram as a guidance in this case.
- The two green vertical lines indicate the first and third quartile of the distribution. Again, the values of the quartiles are listed in the legend.
- The red dotted line indicates a normal distribution defined by the measured mean and the measured standard derivation. The value of the standard derivation is given again in the legend.
- Finally, the legend contains the value for the Shannon Entropy that the measured test sample contains. The Shannon Entropy is calculated with the formula specified in Section 5.2 using the observations after cutting off the outliers above the threshold mentioned above.

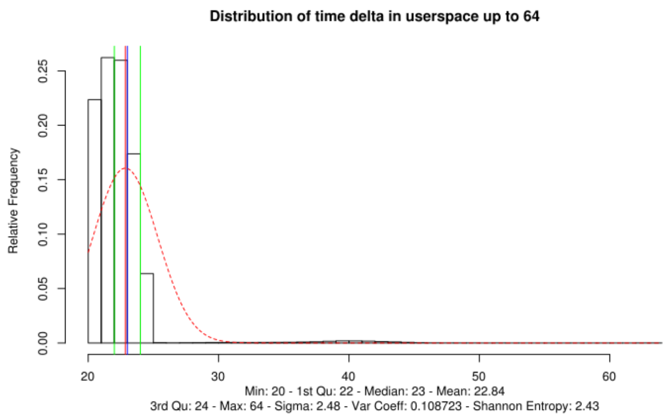


Figure 2: Distribution of time variances in user space over 1.000.000 loops

The graph together with the code now illustrates the variation in execution time of the very same set of operations – it illustrates the CPU execution time jitter for a very tight loop. As these variations are based on the aforementioned complexity of the operating system and its use of hardware mechanisms, no observer can deduce the next variation with full certainty even though the observer is able to fully monitor the operation of the system. And these non-deterministic variations are the foundation of the proposed CPU Jitter random number generator.

As the CPU Jitter random number generator is intended to work in kernel space as well, the same analysis is performed for the kernel. For an initial test, the time stamp variance collection is invoked 30.000.000 times. The generation of the given number of time deltas is very fast, typically less than 10 seconds. When re-performing the test, the distribution varies greatly, including the Shannon Entropy. The lowest observed value was in the 1.3 range and the highest was about 3. The reason for not obtaining a longer sample is simply resources: calculating the graph would take more than 8 GB of RAM.

Now that we have established the basic source of entropy, the subsequent design description of the random number generator must explain the following two aspects which are the basic quality requirements discussed in Section 1 applied to our entropy phenomenon:

1. The random number generator design must be capable of preserving and collecting the entropy from the discussed phenomenon. Thus, the random number generator must be able to "magnify" the entropy phenomenon.

2. The random number generator must use the observed CPU execution time jitter to generate an output bit string that delivers the entropy to a caller. That output string must not show any statistical anomalies that allow an observer to deduce any random numbers or increase the probability when guessing random numbers and thus reducing its entropy.

The following section presents the design of the random number generator. Both requirements will be discussed.

3 Random Number Generator Design

The CPU Jitter random number generator uses the above illustrated operation to read the high-resolution timer for obtaining time stamps. At the same time it performs operations that are subject to the CPU execution time jitter which also impact the time stamp readings.

3.1 Maintenance of Entropy

The heart of the random number generator is illustrated in Figure 3.

The random number generator maintains a 64 bit unsigned integer variable, the entropy pool, that is indicated with the gray shaded boxes in Figure 3 which identify the entropy pool at two different times in the processing.

In a big picture, the random number generator implements an entropy collection loop that

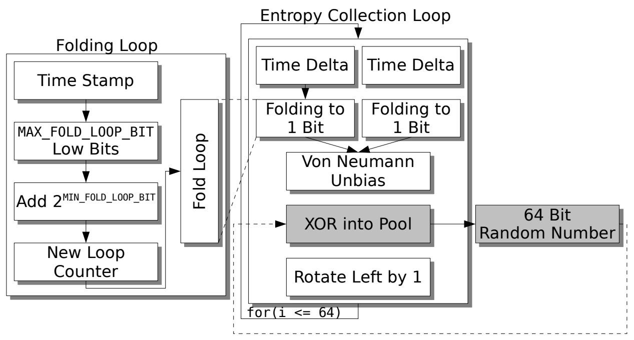


Figure 3: Entropy Collection Operation

1. fetches a time stamp to calculate a delta to the time stamp of the previous loop iteration,

2. folds the time delta value into one bit,

3. processes this value with a Von-Neumann unbias operation,

4. adds this value to the entropy pool using XOR,

5. rotates the pool to fill the next bit value of the pool.

The loop is executed exactly 64 times as each loop iteration generates one bit to fill all 64 bits of the entropy pool. After the loop finishes, the contents of the entropy pool is given to the caller as a 64 bit random number[1]. The following subsection discuss every step in detail.

When considering that the time delta is always computed from the delta to the previous loop iteration, and the fact that the majority of the execution time is spent in the folding loop, the central idea of the CPU Jitter Random Number Generator is to measure the execution time jitter over the execution of the folding loop.

3.1.1 Obtaining Time Delta

The time delta is obtained by:

1. Reading a time stamp,

2. Subtracting that time stamp from the time stamp calculated in the previous loop iteration,

[1]If the caller provides an oversampling rate of greater than 1 during the allocation of the entropy collector, the loop iteration count of 64 is multiplied by this oversampling rate value. For example, an oversample rate of 3 implies that the 64 loop iterations are executed three times – i.e. 192 times.

3. Storing the current time stamp for use in the next loop iteration to calculate the next delta.

For every new request to generate a new random number, the first iteration of the loop is used to "prime" the delta calculation. In essence, all steps of the entropy collection loop are performed, except of mixing the delta into the pool and rotating the pool. This first iteration of the entropy collection loop does not impact the number of iterations used for entropy collection. This is implemented by executing one more loop iteration than specified for the generation of the current random number.

When a new random number is to be calculated, i.e. the entropy collection loop is triggered anew, the previous contents of the entropy pool, which is used as a random number in the previous round is reused. The reusing shall just mix the data in the entropy pool even more. But the implementation does not rely on any properties of that data. The mixing of new time stamps into the entropy pool using XOR ensures that any entropy which may have been left over from the previous entropy collection loop run is still preserved. If no entropy is left, which is the base case in the entropy assessment, the already arbitrary bit pattern in the entropy pool does not negatively affect the addition of new entropy in the current round.

3.1.2 Folding Operation of Time Delta

The folding operation is depicted by the left side of Figure 3. That folding operation is implemented by a loop where the loop counter is not fixed.

To calculate the new fold loop counter a new time stamp is obtained. All bits above the value `MAX_FOLD_LOOP_BITS` – which is set to 4 – are zeroed. The idea is that the fast moving bits of the time stamp value determine the size of the collection loop counter. Why is it set to 4? The 4 low bits define a value between 0 and 16. This uncertainty is used to quickly stabilize the distribution of the output of that folding operation to an equidistribution of 0 and 1, i.e. about 50% of all output is 0 and also about 50% is 1. See Section 5.2.1 for a quantitative analysis of that distribution. To ensure that the collection loop counter has a minimum value, the value 1 is added – that value is controlled with `MIN_FOLD_LOOP_BIT`. Thus, the range of the folding counter value is from 1 to (16 + 1 - 1). Now, this newly determined collection loop counter is used to perform a new fold loop as discussed in the following.

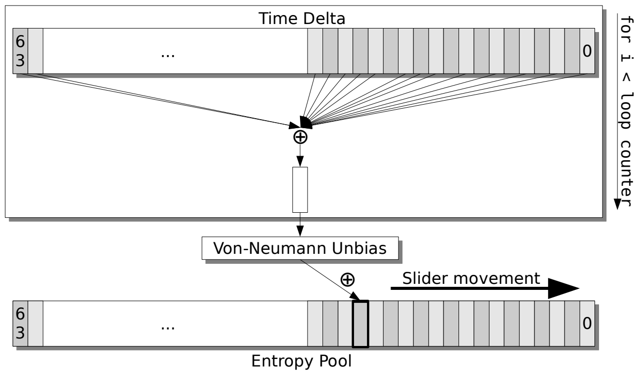


Figure 4: Folding of the time delta and mixing it into the entropy pool

Figure 4 shows the concept of the folding operation of one time delta value.

The upper 64 bit value illustrated in Figure 4 is the time delta obtained at the beginning of the current entropy collection loop iteration. Now, the time delta is partitioned into chunks of 1 bit starting at the lowest bit. The different shades of gray indicate the different 1 bit chunks. The 64 1 bit chunks of the time value are XORed with each other to form a 1 bit value. With the XORing of all 1 bit chunks with each other, any information theoretical entropy that is present in the time stamp will be preserved when folding the value into the 1 bit. But as we fold it into 1 bit, the maximum entropy the time stamp can ever add to the entropy pool is, well, 1 bit. The folding operation is done as often as specified in the loop count.

3.1.3 Von-Neumann Unbias Operation

According to RFC 1750 section 5.2.2, a Von-Neumann unbias operation can be considered to remove any potential skews that may be present in the bit stream of the noise source. The operation is used to ensure that in case skews are present, they are eliminated. The unbias operation is only applicable if the individual consecutive bits are considered independent. Chapter 5 indicates the independence of these individual bits.

To perform the Von-Neumann unbias operation, two independently generated folded bits are processed.

3.1.4 Adding Unbiased Folded Time Delta To Entropy Pool

After obtaining the 1 bit folded and unbiased time stamp, how is it mixed into the entropy pool? The lower 64 bit value in Figure 4 indicates the entropy pool. The 1 bit folded value is XORed with 1 bit from the entropy pool.

But which bit is used? The rotation to the left by 1 bit that concludes the entropy collection loop provides the answer. When the entropy collection loop perform the very first iteration, the 1 bit is XORed into bit 0 of the entropy pool. Now, that pool is rotated left by 1 bit. That means that bit 63 before the rotation becomes bit 0 after the rotation. Thus, the next round of the entropy collection loop XORes the 1 bit folded time stamp again into bit 0 which used to be bit 63 in the last entropy collection loop iteration[2].

The reason why the rotation is done with the value 1 is due to the fact that we have 1 bit we want to add to the pool. The way how the folded bit values are added to the entropy pool can be viewed differently from a mathematical standpoint when considering 64 1 bit values: instead of saying that each of the 64 1 bit value is XORed independently into the entropy pool and the pool value is then rotated, it is equivalent to state that 64 1 bit values are concatenated and then the concatenated value is XORed into the entropy pool. The reader shall keep that analogy in mind as we will need it again in Section 5.

3.2 Generation of Random Number Bit Stream

We now know how one 64 bit random number value is generated. The interface to the CPU Jitter random number generator allows the caller to provide a pointer to memory and a size variable of arbitrary length. The random number generator is herewith requested to generate a bit stream of random numbers of the requested size that is to be stored in the memory pointed to by the caller.

[2]Note, Figure 4 illustrates that the the folded bit of the time delta is moved over the 64 bit entropy pool as indicated with the bold black box (a.k.a the "slider"). Technically, the slider stays at bit 0 and the entropy pool value rotates left. The end result of the mixing of the folded bit into the entropy pool, however, is identical, regardless whether you rotate the entropy pool left or move the slider to the right. To keep the figure illustrative, it indicates the movement of the slider.

The random number generator performs the following sequence of steps to fulfill the request:

1. Check whether the requested size is smaller than 64 bits. If yes, generate one 64 bit random number, copy the requested amount of bits to the target memory and stop processing the request. The unused bits of the random number are not used further. If a new request arrives, a fresh 64 bit random number is generated.
2. If the requested size is larger than 64 bits, generate one random number, copy it to the target. Reduce the requested size by 64 bits and decide now whether the remaining requested bits are larger or smaller than 64 bits and based on the determination, follow either step 1 or step 2.

Mathematically step 2 implements a concatenation of multiple random numbers generated by the random number generator.

3.3 Initialization

The CPU Jitter random number generator is initialized in two main parts. At first, a consuming application must call the `jent_entropy_init(3)` function which validates some basic properties of the time stamp. Only if this validation succeeds, the CPU Jitter random number generator can be used.

The second part can be invoked multiple times. Each invocation results in the instantiation of an independent copy of the CPU Jitter random number generator. This allows a consumer to maintain multiple instances for different purposes. That second part is triggered with the invocation of `jent_entropy_collector_alloc(3)` and implements the following steps:

1. Allocation and zeroing of memory used for the entropy pool and helper variables – `struct rand_data` defines the entropy collector which holds the entropy pool and its auxiliary values.
2. Invoking the entropy collection loop once – this fills the entropy pool with the first random value which is not returned to any caller. The idea is that the entropy pool is initialized with some values other than zero. In addition, this invocation

of the entropy collection loop implies that the entropy collection loop counter value is set to a random value in the allowed range.

3. If FIPS 140-2 is enabled by the calling application, the FIPS 140-2 continuous test is primed by copying the random number generated in step 3 into the comparing value and again triggering the entropy collection loop for a fresh random number.

3.4 Memory Protection

The CPU Jitter random number generator is intended for any consuming application without placing any requirements. As a standard behavior, after completing the caller's request for a random number, i.e. generating the bit stream of arbitrary length, another round of the entropy collection loop is triggered. That invocation shall ensure that the entropy pool is overwritten with a new random value. This prevents a random value returned to the caller and potentially used for sensitive purposes lingering in memory for long time. In case paging starts, the consuming application crashes and dumps core or simply a hacker cracks the application, no traces of even parts of a generated random number will be found in the memory the CPU Jitter random number generator is in charge of.

In case a consumer is deemed to implement a type of memory protection, the flag `CRYPTO_CPU_JITTERENTROPY_SECURE_MEMORY` can be set at compile time. This flag prevents the above mentioned functionality.

Example consumers with memory protection are the kernel, and libgcrypt with its secure memory.

3.5 Locking

The core of the CPU Jitter random number generator implementation does not use any locking. If a user intends to employ the random number generator in an environment with potentially concurrent accesses to the same instance, locking must be implemented. A lock should be taken before any request to the CPU Jitter random number generator is made via its API functions.

Examples for the use of the CPU Jitter random number generator with locks are given in the reference implementations outlined in the appendices.

3.6 FIPS 140-2 Continuous Self Test

If the consuming application enables a FIPS 140-2 compliant mode – which is observable by the CPU Jitter random number generator callback of `jent_fips_enabled` – the FIPS 140-2 mode is enabled.

This mode ensures that the continuous self test is enforced as defined by FIPS 140-2.

3.7 Intended Method of Use

The CPU Jitter random number generator must be compiled without optimizations. The discussion in Section 5.1 supported by Appendix F explains the reason.

The interface discussed in Section 3.2 is implemented such that a caller requesting an arbitrary number of bytes is satisfied. The output can be fed through a whitening function, such as a deterministic random number generator or a hash based cryptographically secure whitening function. The appendix provides various implementations of linking the CPU Jitter random number generator with deterministic random number generators.

However, the output can also be used directly, considering the statistical properties and the entropy behavior assessed in the following chapters. The question, however, is whether this is a wise course of action. Whitening shall help to protect the entropy that is in the pool against observers. This especially a concern if you have a central entropy source that is accessed by multiple users – where a user does not necessarily mean human user or application, since a user or an application may serve multiple purposes and each purpose is one "user". The CPU Jitter random number generator is designed to be instantiated multiple times without degrading the different instances. If a user employs its own private instance of the CPU Jitter random number generator, it may be questionable whether a whitening function would be necessary.

But bottom line: it is a decision that the reader or developer employing the random number generator finally has to make. The implementations offered in the appendices offer the connections to whitening functions. Still, a direct use of the CPU Jitter random number generator is offered as well.

3.8 Programming Dependencies on Operating System

The implementation of the CPU Jitter random number generator only uses the following interfaces from the underlying operating systems. All of them are implemented with wrappers in `jitterentropy-base-{*}.h`. When the used operating system offers these interfaces or a developer replaces them with accordingly, the CPU Jitter random number generator can be compiled on a different operating system or for user and kernel space:

- Time stamp gathering: `jent_get_nstime` must deliver the high resolution time stamp. This function is an architecture dependent function with the following implementations:
  - User space:
    - On Mach systems like MacOS, the function `mach_absolute_time` is used for a high-resolution timer.
    - On AIX, the function `read_real_time` is used for a righ resolution timer.
    - On other POSIX systems, the `clock_gettime` function is available for this operation.
  - Linux kernel space: In the Linux kernel, the `get_cycles` function obtains this information. The directory `arch/` contains various assembler implementations for different CPUs to avoid using an operating system service. If `get_cycles` returns 0, which is possible on several architectures, such as MIPS, the kernel-internal call `__getnstimeofday` is invoked which uses the best available clocksource implementation. The goal with the invocation of `__getnstimeofday` is to have a fallback for `get_cycles` returning zero. Note, if that clocksource clock also is a low resolution timer like the Jiffies timer, the initialization function of the CPU Jitter Random Number Generator is expected to catch this issue.
- `jent_malloc` is a wrapper for the `malloc` function call to obtain memory.
- `jent_free` is a wrapper for calling the `free` function to release the memory.
- `__u64` must be a variable type of a 64 bit unsigned integer – either unsigned long on a 64 bit system or unsigned long long on a 32 bit system.

Loop count	0	1	2	3	4	Bit sum	Figure
1	0	1	1	0	0	N/A	N/A
2	0	0	0	1	0	N/A	N/A
3	1	1	0	0	1	4	5
Result 1	1	2	1	1	1	6	7
Result 2	1	2	1	2	1	7	9

Table 1: Example description of tests

The following additional functions provided by an operating system are used without a wrapper as they are assumed to be present in every operating environment:

- `memcpy`
- `memset`

4 Random Generator Statistical Assessment

After the discussion of the design of the entropy collection, we need to perform assessments of the quality of the random number generator. As indicated in Section 1, the assessment is split into two parts.

This chapter contains the assessment of the statistical properties of the data in the entropy pool and the output data stream.

When compiling the code of the CPU Jitter random number generator with the flag `CRYPTO_CPU_JITTERENTROPY_STAT`, instrumentations are added to the code that obtain the data for the following graphs and distributions. The tests can be automatically reperformed by invoking the `tests_[userspace|kernel]/getstat.sh` shell script which also generates the graphs using the `R-Project` language toolkit.

4.1 Statistical Properties of Entropy Pool

During a testing phase that generated 1,000,000 random numbers, the entropy pool is observed. The observation generated statistical analyses for different aspects illustrated in Table 1. Each line in the table is one observation of the entropy pool value of one round of the entropy collection loop. To read the table, assume that the entropy pool is only 10 bits in size. Further, assume that our entropy collection loop count is 3 to generate a random number.

The left column contains the entropy collection loop count and the indication for the result rows. The middle columns are the 5 bits of the entropy pool. The Bit sum column sums the set bits in the respective row. The Figure column references the figures that illustrate the obtained test data results.

The "Result 1" row holds the number of bits set for each loop count per bit position. In the example above, bit 0 has a bit set only once in all three loops. Bit 1 is set twice. And so on.

The "Result 2" row holds the number of changes of the bits for each loop count compared to the previous loop count per bit position. For example, for bit 0, there is only one change from 0 to 1 between loop count 2 and 3. For bit 7, we have two changes: from 0 to 1 and from 1 to 0.

The graphs contains the same information as explained for Figure 2.

The bit sum of loop count 3 is simply the sum of the set bits holds the number of set bits at the last iteration count to generate one random number. It is expected that this distribution follows a normal distribution closely, because only such a normal distribution is supports implies a rectangular distribution of the probability that each bit is equally likely to be picked when generating a random number output bit stream. Figure 5 contains the distribution of the bit sum for the generated random numbers in user space.

In addition, the kernel space distribution is given in Figure 6 – they are almost identical and thus show the same behavior of the CPU Jitter random number generator

Please note that the black line in the graphs above is an approximation of the density of the measurements using the histogram. When more histogram bars would be used, the approximation would better fit the theoretical normal distribution curve given with the red dotted line. Thus, the difference between both lines is due to the way the graph is drawn and not seen in the actual numbers. This applies also to the bars of the histogram since they are left-aligned which means that on the left

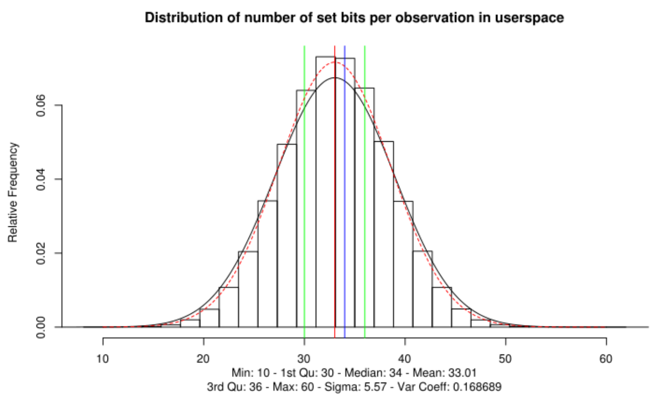


Figure 5: Bit sum of last round of entropy collection loop user space

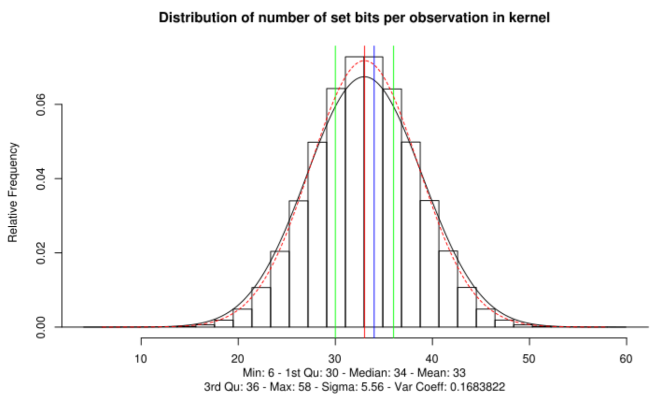


Figure 6: Bit sum of last round of entropy collection loop kernel space

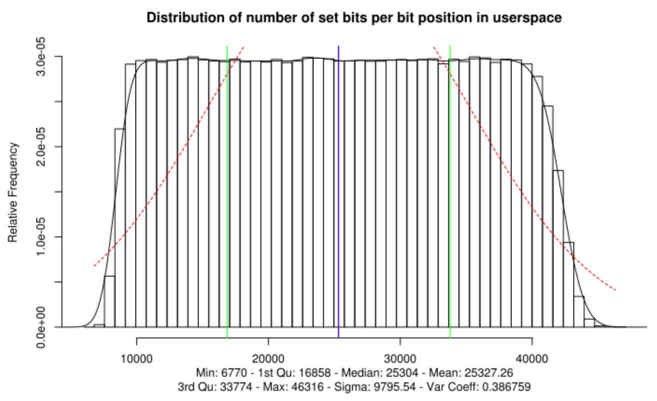


Figure 7: Bit sum of set bits per bit position in user space

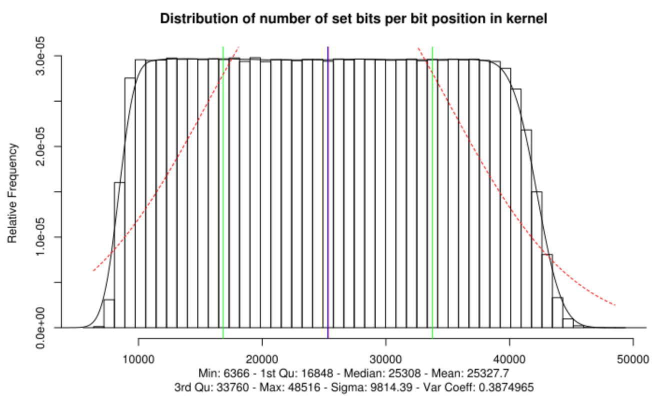


Figure 8: Bit sum of set bits per bit position in kernel space

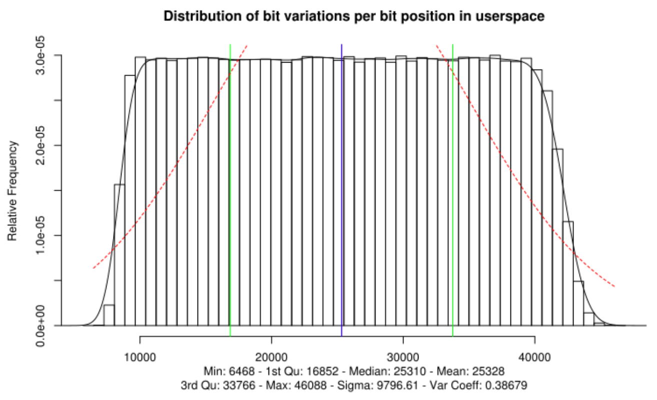


Figure 9: Bit sum of bit variations per bit position in user space

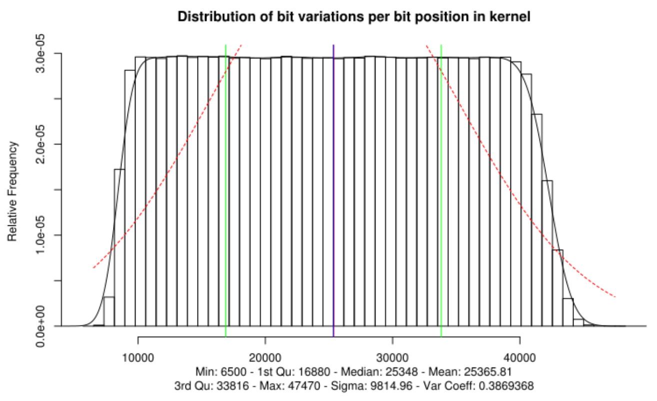


Figure 10: Bit sum of bit variations per bit position in kernel space

side of the diagram they overstep the black line and on the right side they are within the black line.

The distribution for "Result 1" of the sum of of these set bits is given in Figure 7.

Again, for the kernel we have an almost identical distribution shown in Figure 8. And again, we conclude that the behavior of the CPU Jitter random number generator in both worlds is identical.

Just like above, the plot for the kernel space is given in Figure 10.

A question about the shape of the distribution should be raised. One can have no clear expectations about the distribution other than it must show the following properties:

- It is a smooth distribution showing no breaks.
- It is a symmetrical distribution whose symmetry point is the mean.

The distribution for "Result 2" of the sum of of these bit variations in user space is given in Figure 9.

Just like for the preceding diagrams, no material difference is obvious between kernel and user space. The shape of the distributions is similar to the one for the distribution of set bits. An expected distribution can also not be given apart from the aforementioned properties.

4.2 Statistical Properties of Random Number Bit Stream

The discussion of the entropy in Section 5 tries to show that one bit of random number contains one bit of entropy. That is only possible if we have a rectangular distribution of the bits per bit position, i.e. each bit in the output bit stream has an equal probability to be set. The CPU Jitter random number block size is 64 bit. Thus when generating a random number, each of the 64 bits must have an equal chance to be selected by the random number generator. Therefore, when generating large amounts of random numbers and sum the bits per bit position, the resulting distribution must be rectangular. Figure 11 shows the distribution of the bit sums per bit position for a bit stream of 10,000,000 random numbers, i.e 640,000,000 bits.

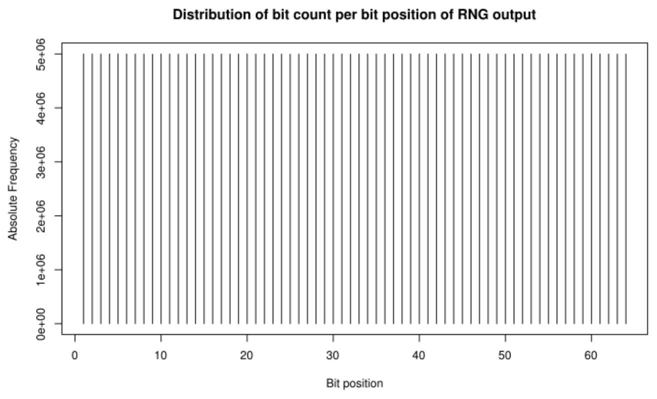


Figure 11: Distribution of bit count per bit position of RNG output

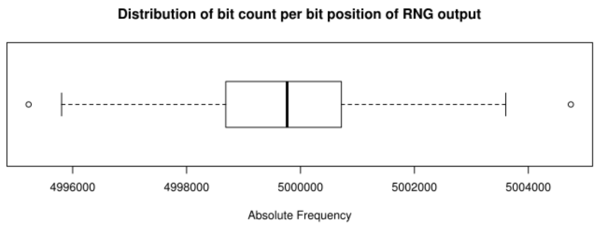


Figure 12: Box plot of variations in bit count per bit position of RNG output

Figure 11 looks pretty rectangular. But can the picture be right with all its 64 vertical lines? We support the picture by printing the box plot in Figure 12 that shows the variance when focusing on the upper end of the columns.

The box plot shows the very narrow fluctuation around expected mean value of half of the count of random numbers produced, i.e. 5,000,000 in our case. Each bit of a random number has the 50% chance to be set in one random number. When looking at multiple random numbers, a bit still has the chance of being set in 50% of all random numbers. The fluctuation is very narrow considering the sample size visible on the scale of the ordinate of Figure 11.

Thus, we conclude that the bit distribution of the random number generator allows the possibility to retain one bit of entropy per bit of random number.

This conclusion is supported by calculating more thorough statistical properties of the random number bit stream are assessed with the following tools:

- `ent`
- `dieharder`
- BSI Test Procedure A

The `ent` tool is given a bit stream consisting of 10,000,000 random numbers (i.e. 80,000,000 Bytes) with the following result where `ent` calculates the statistics when treating the random data as bit stream as well as byte stream:

```
$ dd if=/sys/kernel/debug/jitterentropy/seed of=random.out bs=8 count=10000000

# Byte stream
$ ent random.out
Entropy = 7.999998 bits per byte.

Optimum compression would reduce the size
of this 80000000 byte file by 0 percent.

Chi square distribution for 80000000 samples is 272.04, and randomly
would exceed this value 25.00 percent of the times.

Arithmetic mean value of data bytes is 127.4907 (127.5 = random).
Monte Carlo value for Pi is 3.141600679 (error 0.00 percent).
Serial correlation coefficient is 0.000174 (totally uncorrelated = 0.0).

# Bit stream
$ ent -b random.out
Entropy = 1.000000 bits per bit.

Optimum compression would reduce the size
of this 640000000 bit file by 0 percent.

Chi square distribution for 640000000 samples is 1.48, and randomly
would exceed this value 25.00 percent of the times.

Arithmetic mean value of data bits is 0.5000 (0.5 = random).
Monte Carlo value for Pi is 3.141600679 (error 0.00 percent).
Serial correlation coefficient is -0.000010 (totally uncorrelated = 0.0).
```

During many re-runs of the `ent` test, most of the time, the Chi-Square test showed the test result of 50%, i.e. a perfect result – but even the shown 25% is absolutely in line with random bit pattern. Very similar results were obtained when executing the same test on:

- an Intel Atom Z530 processor;
- a MIPS CPU for an embedded device;
- an Intel Pentium 4 Mobile CPU;
- an AMD Semperon processor;
- KVM guest where the host was based on an Linux 3.8 kernel and with QEMU version 1.4 without any special configuration of hardware access;
- OpenVZ guest on an AMD Opteron processor.
- Fiasco.OC microkernel;

In addition, an unlimited bit stream is generated and fed into `dieharder`. The test results are given with the files `tests_userspace/dieharder-res.*`. The result files demonstrate that all statistical properties tested by `dieharder` are covered appropriately.

The BSI Test Suite A shows no statistical weaknesses.

The test tools indicate that the bit stream complies with the properties of random numbers.

4.3 Anti-Tests

The statistical analysis given above indicates a good quality of the random number generator. To support that argument, an "anti" test is pursued to show that the quality is *not* provided by the post-processing of the time stamp data, but solely by the randomness of the time deltas. The post-processing therefore is only intended to transform the time deltas into a bit string with a random pattern and magnifying the timer entropy.

The following subsections outline different "anti" tests.

4.3.1 Static Increment of Time Stamp

The test is implemented by changing the function `jent_get_nstime` to maintain a simple value that is incremented by 23 every time a time stamp is requested. The value 23 is chosen as it is a prime. Yet, the increment is fully predictable and does not add any entropy.

Analyzing the output bit stream shows that the Chi-Square test of `ent` in both byte-wise and bit-wise output will result in the value of 0.01 / 100.00 which indicates a bit stream that is not random. This is readily clear, because the time delta calculation always returns the same value: 23.

Important remark: The mentioned test can only be conducted when the CPU Jitter random number generator initialization function of `jent_entropy_init(3)` is not called. This function implements a number of statistical tests of the time source. In case the time source would operate in static increments, the initialization function would detect this behavior and return an error.

If the CPU Jitter random number generator would be used with a cryptographic secure whitening function, the outlined "anti" test would *not* show any problems in the output stream. That means that a cryptographic whitening function would hide potential entropy source problems!

4.3.2 Pattern-based Increment of Time Stamp

Contrary to the static increment of the time stamp, this "anti" test describes a pattern-based increment of the time stamp. The time stamp is created by adding the sum of 23 and an additional increment between 1 and 4 using the following code:

```
static unsigned int pad = 0;
static __u64 tmp = 0;
static inline void jent_get_nstime(__u64 *out)
{
        tmp += 23;
        pad++;
        *out = (tmp + (pad & 0x3));
}
```

The code adds 24 in the first loop, 25 in the second, 26 in the third, 27 in the fourth, again 24 in the fifth, and so forth.

Using such a pattern would again fail the `ent` test as the Chi-Square test is at 100 or 0.01 and the data stream can be compressed. Thus, such a time stamp increment would again be visible in the statistical analysis of this chapter.

In addition to the Chi-Square test, the measurements of the second derivation of the time stamp, the variations of time deltas, would present very strange patterns like, zero, or spikes, but no continuously falling graph as measured.

4.3.3 Disabling of System Features

The CPU jitter is based on properties of the system, such as caches. Some of these properties can be disabled in either user space or kernel space. The effect on such changes is measured in various tests.

5 Entropy Behavior

As the previous chapter covered the statistical properties of the CPU Jitter random number generator, this chapter provides the assessment of the entropy behavior. With this chapter, the second vital aspect of random number generators mentioned in Section 1 is addressed.

The CPU Jitter random number generator does not maintain any entropy estimator. Nor does the random number generator tries to determine the entropy of the individual recorded time deltas that are fed into the entropy pool. There is only one basic rule that the CPU Jitter random number generator follows: upon completion of the entropy collection loop, the entropy pool contains 64 bit of entropy which are returned to the caller. That results in the basic conclusion of the random number bit stream returned from the CPU Jitter random number generator holding one bit of entropy per bit of random number.

Now you may say, that is a nice statement, but show me the numbers. The following sections will demonstrate the appropriateness of this statement.

Section 5.1 explains the base source of entropy for the CPU Jitter random number generator. This section explains how the root cause of entropy is visible in the CPU Jitter random number generator. With Section 5.2, the explanation is given how the entropy that is present in the root cause, the CPU execution time jitter, is harvested, maintained through the processing of the random number generator and accumulated in the entropy pool. This section provides the information theoretical background to back up the statistical analyses given in Section 4.

Before we start with the entropy discussion, please let us make one issue perfectly clear: the nature of entropy, which is an indication of the level of uncertainty present in a set of information, can per definition *not* be calculated. All what we can do is try to find arguments whether the entropy estimation the CPU Jitter random number generator applies is valid. Measurements are used to support that assessment. Moreover, the discussion must contain a worst case analysis which gives a lower boundary of the entropy assumed to be present in the random number bit stream extracted from the CPU Jitter random number generator.

5.1 Base Entropy Source

As outlined in Section 3, the variations of the time delta is the source of entropy. Unlike the graphs outlined in Section 2 where two time stamps are invoked immediately after each other, the CPU Jitter random number generator places the folding loop between each time stamp gathering. That implies that the CPU jitter over the folding loop is measured and used as a basis for entropy.

Considering the fact that the CPU execution time jitter over the folding loop is the source of entropy, we can determine the following:

- The result of the folding loop shall return a one bit value that has one bit of entropy.
- The delta of two time stamps before and after the folding loop is given to the folding loop to obtain the one bit value.

When viewing both findings together, we can conclude that the CPU jitter of the time deltas each folding loop shows *must* exceed 1 bit of entropy. Only this way we can ensure that the folded time delta value has one bit of entropy – see Section 5.2.1 for an explanation why the folding operation retains the entropy present in the time delta up to one bit.

Tests are implemented that measure the variations of the time delta over an invocation of the folding loop. The tests are provided with the `tests_userspace/timing/jitterentropy-foldtime.c` test case for user space, and the `stat-fold` DebugFS file for testing the kernel space. To ensure that the measurements are based on the worst-case analysis, the user space test is compiled with `-O2` optimization[3]. The kernel space test is compiled with the same optimization as the kernel itself.

The design of the folding loop in Section 3.1.2 explains that the number of folding loop iterations varies between 2^0 and 2^4 iterations. The testing of the entropy of the folding loop must identify the lower boundary and the upper boundary. The lower boundary is the minimum entropy the folding loop at least will have: this minimum entropy is the entropy observable over a fixed folding loop count. The test uses 2^0 as the fixed folding loop

[3]The CPU execution time jitter varies between optimized and non-optimized binaries. Optimitzed binaries show a smaller jitter compared to non-optimized binaries. Thus, the test applies a worst case approach with respect to the optimizations, even though the design requires the compilation without optimizations.

count. On the other hand, the upper boundary of the entropy is set by allowing the folding loop count to float freely within the above mentioned range.

It is expected that the time stamps used to calculate the folding loop count is independent from each other. Therefore, the entropy observable with the testing of the upper boundary is expected to identify the entropy of the CPU execution time jitter. Nonetheless, if the reader questions the independence, the reader must conclude that the real entropy falls within the measured range between the lower and upper boundary.

Figure 13 presents the lower boundary of the folding loop executing in user space of the test system. The graph shows two peaks whereas the higher peak is centered around the execution time when the code is in the CPU cache. For the time when the code is not in the CPU cache – such as during context switches or during the initial invocations – the average execution time is larger with the center at the second peak. In addition, Figure 14 provides the upper boundary of the folding loop. With the graph of the upper boundary, we see 16 spikes which are the spikes of the lower boundary scattered by the folding loop counter. If the folding loop counter is 1, the variation of the time delta is centered around a lower value than the variations of a folding loop counter of 2 and so on. As the variations of the delta are smaller than the differences between the means of the different distributions, we observe the spikes.

The two graphs use the time deltas of 10,000,000 invocations of the folding loop. To eliminate outliers, time delta values above the number outlined in the graphs are simply cut off. That means, when using all values of the time delta variations, the calculated Shannon Entropy would be higher than listed in the legend of the graphs. This cutting off therefore is yet again driven by the consideration of determining the worst case.

The lower boundary shows a Shannon Entropy above 2.9 bits and the upper boundary a Shannon Entropy above 6.7 bits.

In addition to the user space measurements, Figures 15 and 16 present the lower and upper boundary of the folding loop execution time variations in kernel space on the same system. Again, the lower boundary is above 2 bits and the upper above 6 bits of Shannon Entropy.

As this measurement is the basis of all entropy discussion, Appendix F shows the measurements for many

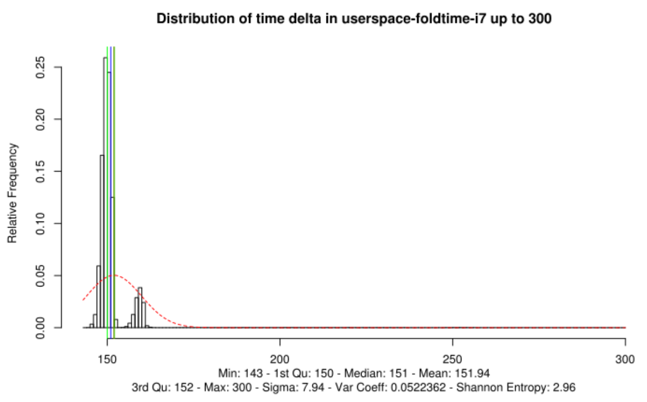


Figure 13: Lower boundary of entropy over folding loop in user space

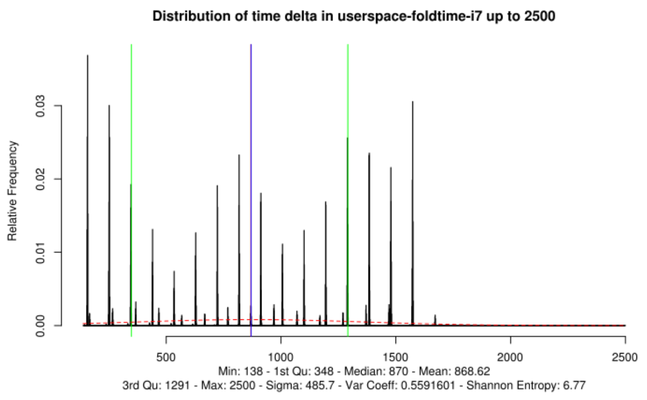


Figure 14: Upper boundary of entropy over folding loop in user space

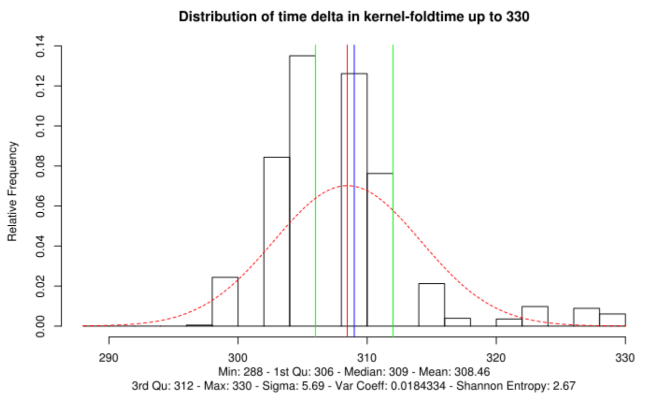


Figure 15: Lower boundary of entropy over folding loop in kernel space

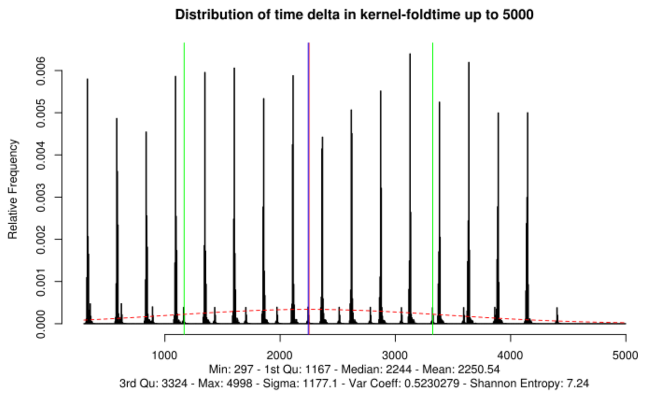


Figure 16: Upper boundary of entropy over folding loop in kernel space

different CPUs. All of these measurements show that the lower and upper boundaries are always much higher than the required one bit of entropy with exceptions. All tests are executed with optimized code as even a worst case assessment and sometimes with the non-optimized compilation to show the difference.

For the other CPUs whose lower entropy is below 1 bit and the `jent_entropy_init` function allows this CPU, statistical tests are performed to verify that no cycles are present. This implies that the entropy is closer to the upper boundary and therefore well above 1 bit.

The reader should also consider that the measured Shannon Entropy is a conservative measurement as the test invokes the folding loop millions of times successively. This implies that for the entire duration of the test, caches, branch prediction units and similar are mostly filled with the test code and thus have hardly any impact on the variations of the time deltas. In addition, the test systems are kept idle as much as possible to limit the number of context switches which would have an impact on the cache hits. In real-life scenarios, the caches are typically filled with information that have an big impact on the jitter measurements and thus increase the entropy.

With these measurements, we can conclude that the CPU execution jitter over the folding loop is always more than double the entropy in the worst case than required. Thus, the measured entropy of the CPU execution time jitter that is the basis of the CPU Jitter random number generator is much higher than required.

The reader may now object and say that the measured values for the Shannon Entropy are not appropriate for the real entropy of the execution time jitter, because the observed values may present some patterns. Such patterns would imply that the real entropy is significantly lower than the calculated Shannon Entropy. This argument can easily be refuted by the statistical tests performed in Section 4. If patterns would occur, some of the statistical tests would indicate problems. Specifically the Chi-Square test is very sensitive to any patterns. Moreover, the "anti" tests presented in Section 4.3 explain that patterns are easily identifiable.

5.1.1 Impact of Frequency Scaling and Power Management on Execution Jitter

When measuring the execution time jitter on a system with a number of processes active such as a system with the X11 environment and KDE active, one can identify that the absolute numbers of the execution time of a folding loop is higher at the beginning than throughout the measurement. The behavior of the jitter over time is therefore an interesting topic. The following graph plots the first 100,000 measurements[4] where all measurements of time deltas above 600 were removed to make the graph more readable (i.e. the outliers are removed). It is interesting to see that the execution time has a downward trend that stabilizes after some 60,000 folding loops. The downward trend, however, is not continuously but occurs in steps. The cause for this behavior is the frequency scaling (Intel SpeedStep) and power management of the system. Over time, the CPU scales up to the maximum processing power. Regardless of the CPU processing power level, the most important aspect is that the oscillation within each step has an similar "width" of about 5 to 10 cycles. Therefore, regardless of the stepping of the execution time, the jitter is present with an equal amount! Thus, frequency scaling and power management does not alter the jitter.

When "zooming" in into the graph at different locations, as done below, the case is clear that the oscillation within each step remains at a similar level.

The constant variations support the case that the CPU execution time jitter is agnostic of the with frequency scaling and power management levels.

[4]The measurements of the folding loop execution time were re-performed on the same system that is used for Section 5.1. As the measurements were re-performed, the absolute numbers vary slightly to the ones in the previous section.

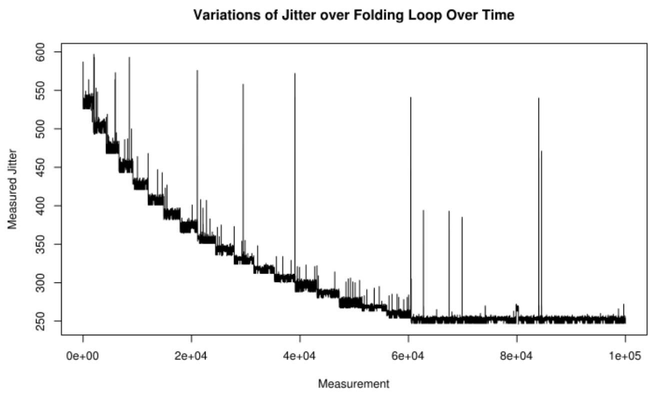


Figure 17: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management

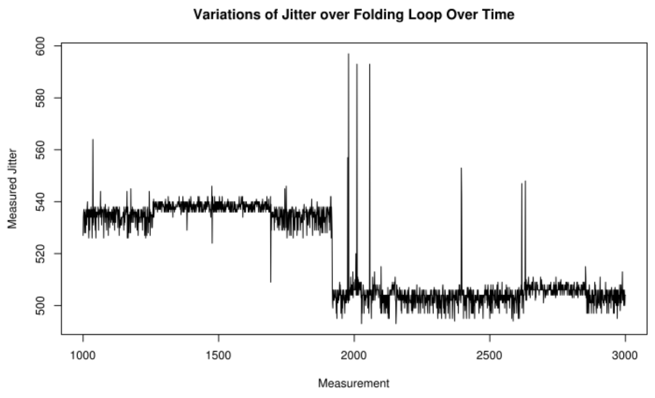


Figure 18: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management – “zoomed in at measurements 1,000 - 3,000”

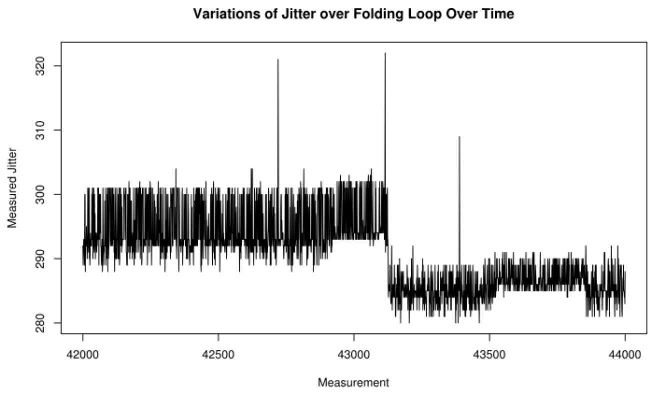


Figure 19: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management – “zoomed in at measurements 42,000 - 44,000”

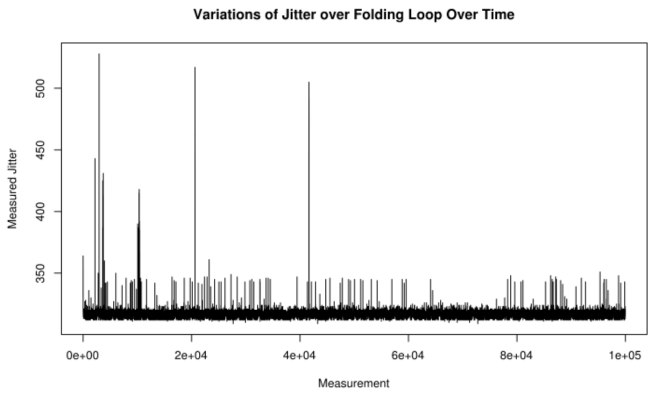


Figure 20: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled

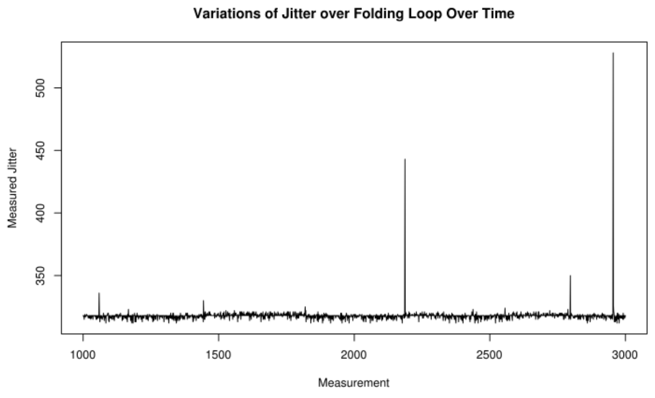


Figure 21: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled – “zoomed in at measurements 1,000 - 3,000”

To compare the measurements with disabled frequency scaling and power management on the same system, the following graphs are prepared. These graphs show the same testing performed.

5.2 Flow of Entropy

Entropy is a is a phenomenon that is typically characterized with the formula for the Shannon Entropy H

$$H = -\sum_{i=1}^{N} p_i \cdot log_2(p_i)$$

where N is the number of samples, and p_i is the probability of sample i. As the Shannon Entropy formula uses

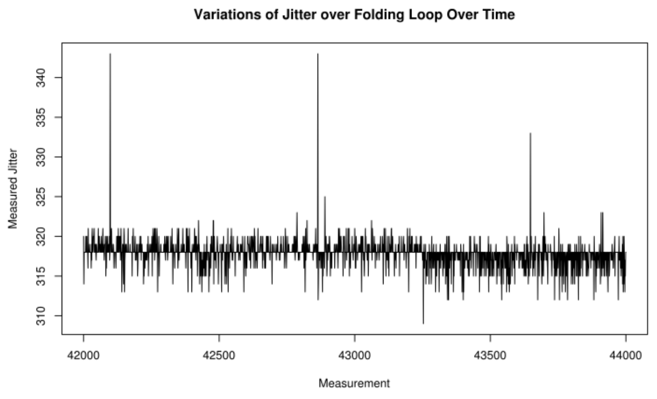


Figure 22: Variations of the execution time jitter over time when performing folding loop jitter measurements with Frequency Scaling / Power Management disabled – "zoomed in at measurements 42,000 - 44,000"

the logarithm at base 2, that formula results in a number of bits of entropy present in an observed sample.

Considering the logarithm in the Shannon Entropy formula one has to be careful on which operations can be applied to data believed to contain entropy to not lose it. The following operations are allowed with the following properties:

- Concatenation of bit strings holding entropy implies that the combined string contains the combination of both entropies, i.e. the entropy value of both strings are added. That is only allowed when both observations are independent from each other.
- A combination of the bit strings of two *independent* observations using XOR implies that the resulting string holds the entropy equaling to larger entropy of both strings – for example XORing two strings, one string with 10 bits in size and 5 bits of entropy and another with 20 bits holding 2 bits results in a 20 bit string holding 5 bits of entropy. The key is that even a string with 0 entropy XORed with a string holding entropy will not diminish the entropy of the latter.

Any other operation, including partial overlapping concatenation of strings will diminish the entropy in the resulting string in ways that are not easily to be determined. These properties set the limit in which the CPU Jitter random number generator can process the time stamps into a random bit stream.

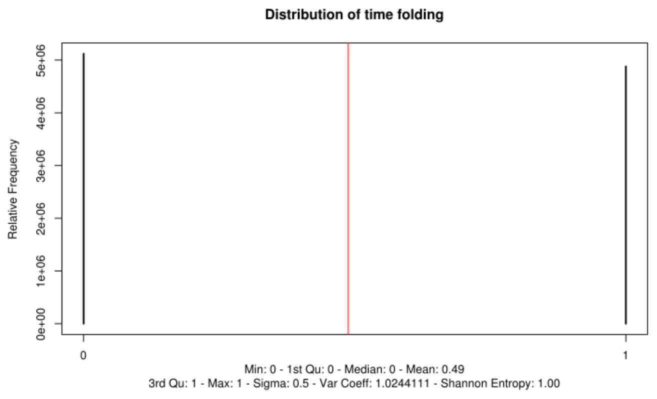


Figure 23: Measurement of time folding operation

The graphs about the distribution of time deltas and their variations in Section 5.1 include an indication of the Shannon Entropy which is based on the observed samples using the mentioned formula for the Shannon Entropy. In each case, the Shannon Entropy is way above 1 bit – a value which is fundamental to the following discussion.

5.2.1 First Operation: Folding of Time Delta

According to the implementation illustrated with Figure 3, the first operation after the CPU Jitter random number generator obtains a time delta is the folding operation. The list of allowed operations include the XOR operation. The folding is an XOR operation of the 64 1 bit slices of the 64 bit time stamp. The XOR operation does not diminish the entropy of the overall time stamp when considered as slices. The overall time delta is expected to have more than 1 bit of entropy according to figures in Section 5.1. The string size after the folding is 1 bit and can thus not hold more than 1 bit of entropy.

To measure that entropy, the folding operation is closely analyzed with the test `tests_userspace/timing/jitterentropy-folding.c`. This test performs the folding operation as illustrated in the left hand side of Figure 3, i.e. a time delta is created which is folded. The folded value is recorded and a folding operation is performed. The distribution of the bit value – an integer ranging from 0 to 1 – resulting from the folding operation is recorded. Figure 23 shows the distribution of this test when measuring 10,000,000 invocations of that time stamp with the folding operation applied.

The distribution shows that both values have an equal chance of being selected. That implies that the Shannon Entropy is 1.0 as recorded in the legend of the diagram. We conclude that the folding operation will retain 1 bit of entropy provided that the input, i.e. the timing value holds 1 or more bits of entropy.

Note, the repetition of the folding loop is of no harm to the entropy as the same value is calculated during each folding loop execution.

5.2.2 Second Operation: Von-Neumann Unbias

The Von-Neumann unbias operation does not have an effect on the entropy of the source. The mathematical proof is given in the document A proposal for: Functionality classes for random number generators Version 2.0 by Werner Schindler section 5.4.1 issued by the German BSI.

The requirement on using the Von-Neumann unbias operation rests on the fact that the input to the unbias operation are two independent bits. The independence is established by the following facts:

1. The bit value is determined by the delta value which is affected by the CPU execution jitter. That jitter is considered independent of the CPU operation before the time delta measurement,
2. The delta value is calculated to the previous execution loop iteration. That means that two loop iterations generate deltas based on each individual loop. The delta of the first loop operation is neither part of the delta of the second loop (e.g. when the second delta would measure the time delta of both loop iterations), nor is the delta of the second loop iteration affected by the first operation based on the finding in bullet 1.

5.2.3 Third Operation: Entropy Pool Update

What is the next operation? Let us look again at Figure 3. The next step after folding and unbiasing is the mixing of the folded value into the entropy pool by XORing it into the pool and rotating the pool.

The reader now may say, these are two distinct operations. However, in Section 3.1.2 we already concluded that the XOR operation using 64 1 bit folded values together with the rotation by 1 bit of the entropy pool can mathematically be interpreted as a concatenation of 64 1 bit folded values into a 64 bit string. Thus, both operations are assessed as a concatenation of the individual folded bits into a 64 bit string followed by an XOR of that string into the entropy pool.

Going back to the above mentioned allowed operations with bit strings holding entropy, the concatenation operation adds the entropy of the individual bit strings that are concatenated. Thus, we conclude that the concatenation of 64 strings holding 1 bit of entropy will result in a bit string holding 64 bit of entropy.

When concatenating additional n 1 bit strings into the 64 bit entropy pool will not increase the entropy any more as the rotation operation rolls around the 64 bit value and starts at the beginning of that value again. When the entropy collection loop counter has a value that is not divisible by 64, the last bit string XORed into the entropy pool is less than 64 bits – for example, the counter has the value 260, the 4 last folded bits generated by the loop will form a 4 bit string that is XORed into the entropy pool. This last bit string naturally contains less than 64 bits of entropy – the maximum entropy it contains is equal to the number of bits in that string. Considering the calculation rules for entropy mentioned above, XORing a string holding less entropy with a string with more entropy will not diminish the entropy of the latter. Thus, the XORing of the last bits into the entropy pool will have no effect on the entropy of the entropy pool.

There is a catch to the calculation: the math only applies when the individual observations, i.e. the individual 1 bit folded time delta values, are independent from each other. The argument supporting the independence of the individual time deltas comes back to the fundamental property of the CPU execution time jitter which has an unpredictable variation. Supportive is the finding that one entropy collection loop iteration, which generates a 1 bit folded value, has a much wider distribution compared to Figure 2 – the reader may particularly consider the standard deviation. This variation in the execution time of the loop iteration therefore breaks any potentially present dependencies between adjacent loop counts and their time deltas. Note again, the time deltas we collect only need 1 bit of entropy. Looking at Figure 24 which depicts the distribution of the execution time of one entropy loop iteration, we see that the variation and its included Shannon Entropy is high enough

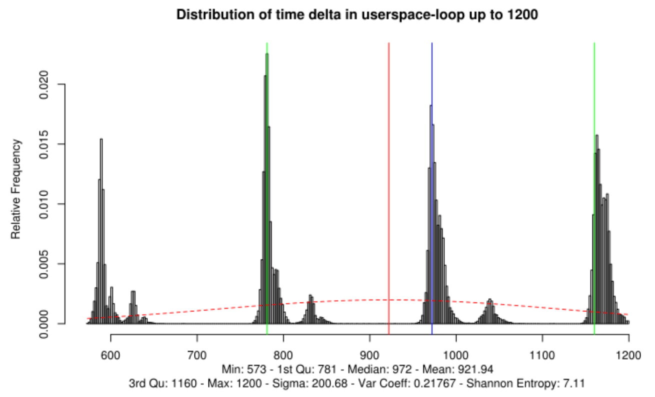


Figure 24: Distribution of execution time of one entropy collection loop iteration

to support the conclusion of an independence between time deltas of adjacent loop iterations.

Thus, we conclude that our entropy pool holds 64 bit of entropy after the conclusion of the mixing operation.

5.2.4 Fourth Operation: Generation of Output String

The fourth and last operation on the bit string holding entropy is the generation of the string of arbitrary length.

The generation of the output string is performed by concatenating random numbers of the size of 64 bit with each other until the resulting bit string matches the requested size. The individual random numbers are generated by independent invocations of the entropy collection loop.

Using concatenation and the conclusion from the preceding sections[5], the entropy in the resulting bit string is equal to the number of bits in that string.

The CPU Jitter random number generator operates on 64 bit blocks – the length of the entropy pool. When the requested bit string length is not divisible by 64 bits, the last chunk concatenated with the output bit stream is therefore less than 64 bits with the reminding bits not given to the caller – note, the caller is only able to specify the output size in bytes and thus in 8-bit chunks. Why is this operation not considered to diminish the entropy of the last chunk below its number of bits? To find the answer, let us go back how the entropy pool is constructed: one bit of folded timer value known to have one bit of entropy is added to the pool. When considering the entropy pool as 64 segments of individual bits, every individual bit still contains 1 bit of entropy, because the only operation each single bit is modified with, is XOR. Thus, every bit in the bit string of the entropy pool holds one bit of entropy. This ultimately implies that when taking a subset of the entropy pool, that subset still has as much entropy as the size of the subset in bits.

[5]The entropy pool contains 64 bit of entropy after the completion of the random number generation.

5.3 Reasons for Chosen Values

The reader now may ask why the time delta is folded into one bit and not into 2 or even 4 bits. Using larger bit strings would reduce the number of foldings and thus speed up the entropy collection. Measurements have shown that the speed of the CPU Jitter random number generator is cut by about 40% when using 4 bits versus 2 bits or 2 bits versus 1 bit. However, the entire argumentation for entropy is based on the entropy observed in the execution time jitter illustrated in Section 5.1. The figures in this section support the conclusion that the Shannon Entropy measured in Section 5.1 is the absolute worst case. To be on the save side, the lower boundary of the measured entropy shall always be significantly higher than the entropy required for the value returned by the folding operation.

Another consideration for the size of the folded time stamp is important: the implications of the last paragraph in Section 5.2.4. The arguments and conclusions in that paragraph only apply when using a size of the folded time stamp that is less or equal 8 bits, i.e. one byte.

6 Conclusion

For the conclusion, we need to get back to Section 1 and consider the initial goals we have set out.

First, let us have a look at the general statistical and entropy requirements. Chapter 4 concludes that the statistical properties of the random number bit stream generated by the CPU Jitter random number generator meets all expectations. Chapter 5 explains the entropy behavior and concludes that the collected entropy by the CPU execution time jitter is much larger than the entropy pool. In addition, that section determines that the way data is mixed into the entropy pool does not diminish the gathered entropy. Therefore, this chapter concludes that one bit of output of the CPU Jitter random number generator holds one bit of information theoretical entropy.

In addition to these general goals, Section 1 lists a number of special goals. These goals are considered to be covered. A detailed assessment on the coverage of these goals is given in the original document.

A Availability of Source Code

The source code of the CPU Jitter entropy random number generator including the documentation is available at http://www.chronox.de/jent/jitterentropy-current.tar.bz2.

The source code for the test cases and R-project files to generate the graphs is available at the same web site.

B Linux Kernel Implementation

The document describes in Section 1 the goals of the CPU Jitter random number generator. One of the goals is to provide individual instances to each consumer of entropy. One of the consumers are users inside the Linux kernel.

As described above, the output of the CPU Jitter random number generator is not intended to be used directly. Instead, the output shall be used as a seed for either a whitening function or a deterministic random number generator. The Linux kernel support provided with the CPU Jitter random number generator chooses the latter approach by using the ANSI X9.31 DRNG that is provided by the Linux kernel crypto API.

Figure 25 illustrates the connection between the entropy collection and the deterministic random number generators offered by the Linux kernel support. The interfaces at the lower part of the illustration indicate the Linux kernel crypto API names of the respective deterministic random number generators and the file names within `/sys/kernel/debug`, respectively.

Every deterministic random number generator instance is seeded with its own instance of the CPU Jitter random number generator. This implementation thus uses one of the design goals outlined in Section 1, namely multiple,

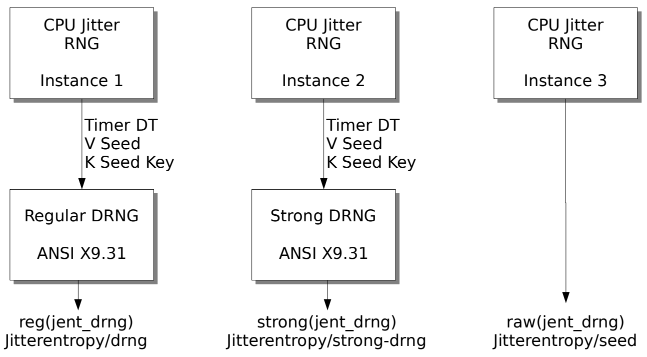

CPU Jitter RNG Instance 1
CPU Jitter RNG Instance 2
CPU Jitter RNG Instance 3
Timer DT
V Seed
K Seed Key
Timer DT
V Seed
K Seed Key
Regular DRNG ANSI X9.31
Strong DRNG ANSI X9.31
reg(jent_drng) Jitterentropy/drng
strong(jent_drng) Jitterentropy/strong-drng
raw(jent_drng) Jitterentropy/seed

Figure 25: Using CPU Jitter RNG to seed ANSI X9.31 DRNGs

unrelated instantiations of the CPU Jitter random number generator.

The offered deterministic random number generators have the following characteristics:

- The regular deterministic random number generator is re-seeded with entropy from the CPU Jitter random number generator after obtaining `MAX_BYTES_RESEED` bytes since the last re-seed. Currently that value is set to 1 kilobytes. In addition, when reaching the limit of `MAX_BYTES_REKEY` bytes since the last re-key, the deterministic random number generator is re-keyed using entropy from the CPU Jitter random number generator. This value is currently set to 1 megabytes.

- The strong deterministic random number generator is re-seeded and re-keyed after the generator of `MAX_BYTES_STRONG_RESEED` bytes and `MAX_BYTES_STRONG_REKEY` bytes, respectively. The re-seeding value is set to 16 bytes, which is equal to the block size of the deterministic random number generator. This implies that the information theoretical entropy of one block of random number generated from the deterministic random number generator is always 16 bytes. The re-key value is set to 1 kilobytes.

- Direct access to the CPU Jitter random number generator is provided to the caller when raw entropy is requested.

Currently, the kernel crypto API only implements a full reset of the deterministic random number generators.

Therefore, the description given above is the plan after the kernel crypto API has been extended. Currently, when hitting the re-seed threshold, the deterministic random number generator is reset with 48 bytes of entropy from the CPU Jitter random number generator. The re-key value is currently not enforced.

B.1 Kernel Crypto API Interface

When compiling the source code with the configuration option `CRYPTO_CPU_JITTERENTROPY_KCAPI`, the kernel crypto API bonding code is compiled. That code registers the mentioned deterministic random number generators with the kernel crypto API. The bonding code provides a very thin wrapper around the management code for the provided random number generators.

The deterministic random number generators connected with as well as the direct access to the CPU Jitter random number generator are accessible using the following kernel crypto API names:

reg(jent_rng) Regular deterministic random number generator

strong(jent_rng) Strong deterministic random number generator

raw(jent_rng) Direct access to the CPU Jitter random number generator which returns unmodified data from the entropy collection loop.

When invoking a reset operation on one of the deterministic random number generator, the implementation performs the re-seed and re-key operations mentioned above on this deterministic random number generator irrespectively whether the thresholds are hit.

A reset on the `raw(jent_rng)` instance is a noop.

B.2 Kernel DebugFS Interface

The kernel DebugFS interface offered with the code is *only* intended for debugging and testing purposes. During regular operation, that code *shall not* be compiled as it allows access to the internals of the random number generation process.

The DebugFS interface is compiled when enabling the `CRYPTO_CPU_JITTERENTROPY_DBG` configuration option. The interface registers the following files within the directory of `/sys/kernel/debug/jitterentropy`:

stat The `stat` file offers statistical data about the regular and strong random number generators, in particular the total number of generated bytes and the number of re-seeds and re-keys.

stat-timer This file contains the statistical timer data for one entropy collection loop count: time delta, delta of time deltas and the entropy collection loop counter value. This data forms the basis of the discussion in Section 4. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

stat-bits This file contains the three tests of the bit distribution for the graphs in Section 4. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

stat-fold This file provides the information for the entropy tests of the folding loop as outlined in Section 5.1. Reading the file will return an error if the code is not compiled with `CRYPTO_CPU_JITTERENTROPY_STAT`.

drng The `drng` file offers access to the regular deterministic random number generator to pull random number bit streams of arbitrary length. Multiple applications calling at the same time are supported due to locking.

strong-rng The `strong-drng` file offers access to the strong deterministic random number generator to pull random number bit streams of arbitrary length. Multiple applications calling at the same time are supported due to locking.

seed The `seed` file allows direct access to the CPU Jitter random number generator to pull random number bit streams of arbitrary lengths. Multiple applications calling at the same time are supported due to locking.

timer The `timer` file provides access to the time stamp kernel code discussed in Section 2. Be careful when obtaining data for analysis out of this file: redirecting the output immediately into a file (even a file on a TmpFS) significantly enlarges the measurement and thus make it look having more entropy than it has.

collection_loop_count This file allows access to the entropy collection loop counter. As this counter value is considered to be a sensitive parameter, this file will return -1 unless the entire code is compiled with the CRYPTO_CPU_JITTERENTROPY_STAT flag. *This flag is considered to be dangerous* for normal operations as it allows access to sensitive data of the entropy pool that shall not be accessible in regular operation – if an observer can access that data, the CPU Jitter random number generator must be considered to deliver much diminished entropy. Nonetheless, this flag is needed to obtain the data that forms the basis of some graphs given above.

B.3 Integration with random.c

The CPU Jitter random number generator can also be integrated with the Linux /dev/random and /dev/urandom code base to serve as a new entropy source. The provided patch instantiates an independent copy of an entropy collector for each entropy pool. Entropy from the CPU Jitter random number generator is only obtained if the entropy estimator indicates that there is no entropy left in the entropy pool.

This implies that the currently available entropy sources have precedence. But in an environment with limited entropy from the default entropy sources, the CPU Jitter random number generator provides entropy that may prevent /dev/random from blocking.

The CPU Jitter random number generator is only activated, if jent_entropy_init passes.

B.4 Test Cases

The directory tests_kernel/kcapi-testmod/ contains a kernel module that tests whether the Linux Kernel crypto API integration works. It logs its information at the kernel log.

The testing of the interfaces exported by DebugFS can be performed manually on the command line by using the tool dd with the files seed, drng, strong-drng, and timer as dd allows you to set the block size precisely (unlike cat). The other files can be read using cat.

C Libgcrypt Implementation

Support to plug the CPU Jitter random number generator into libgcrypt is provided. The approach is to add the callback to the CPU Jitter random number generator into _gcry_rndlinux_gather_random. Thus, the CPU Jitter random number generator has the ability to run every time entropy is requested. Figure 26 illustrates how the CPU Jitter random number generator hooks into the libgcrypt seeding framework.

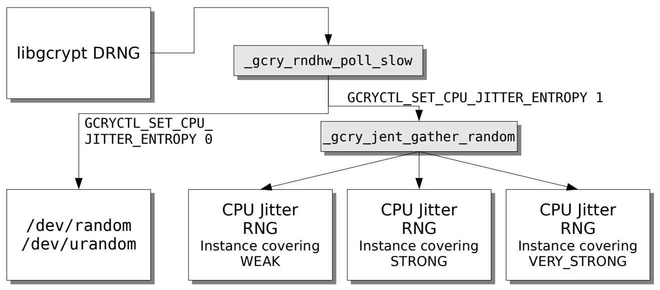

libgcrypt DRNG
_gcry_rndhw_poll_slow
GCRYCTL_SET_CPU_JITTER_ENTROPY 1
GCRYCTL_SET_CPU_JITTER_ENTROPY 0
_gcry_jent_gather_random
/dev/random /dev/urandom
CPU Jitter RNG Instance covering WEAK
CPU Jitter RNG Instance covering STRONG
CPU Jitter RNG Instance covering VERY_STRONG

Figure 26: Use of CPU Jitter RNG by libgcrypt

The wrapper code around the CPU Jitter random number generator provided for libgcrypt holds the following instances of the random number generator. Note, the operation of the CPU Jitter random number generator is unchanged for each type. The goal of that approach shall ensure that each type of seed request is handled by a separate and independent instance of the CPU Jitter random number generator.

weak_entropy_collector Used when GCRY_WEAK_RANDOM random data is requested.

strong_entropy_collector Used when GCRY_STRONG_RANDOM random data is requested.

very_strong_entropy_collector Used when GCRY_VERY_STRONG_RANDOM random data is requested.

The CPU Jitter random number generator with its above mentioned instances is initialized when the caller uses GCRYCTL_SET_CPU_JITTER_ENTROPY with the flag 1. At this point, memory is allocated.

Only if the above mentioned instances are allocated, the wrapper code uses them! That means the callback from _gcry_rndlinux_gather_random to the CPU Jitter random number generator only returns random bytes when these instances are allocated. In turn, if they are not allocated, the normal processing of _gcry_rndlinux_gather_random is continued.

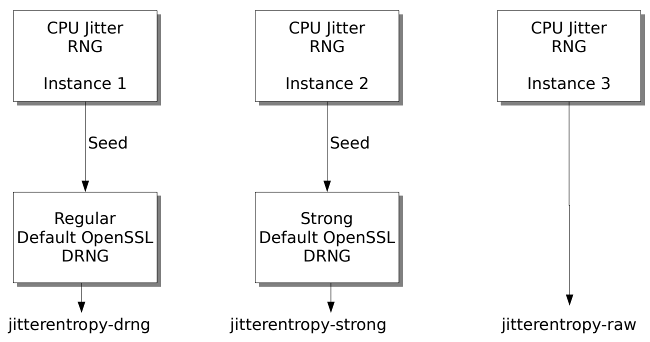

CPU Jitter RNG Instance 1
CPU Jitter RNG Instance 2
CPU Jitter RNG Instance 3
Seed
Seed
Regular Default OpenSSL DRNG
Strong Default OpenSSL DRNG
jitterentropy-drng
jitterentropy-strong
jitterentropy-raw

Figure 27: CPU Jitter random number generator seeding OpenSSL default DRNG

If the user wants to disable the use of the CPU Jitter random number generator, a call to `GCRYCTL_SET_CPU_JITTER_ENTROPY` with the flag 0 must be made. That call deallocates the random number generator instances.

The code is tested with the test application `tests_userspace/libgcrypt/jent_test.c`. When using `strace` on this application, one can see that after disabling the CPU Jitter random number generator, `/dev/random` is opened and data is read. That implies that the standard code for seeding is invoked.

See `patches/README` for details on how to apply the code to libgcrypt.

D OpenSSL Implementation

Code to link the CPU Jitter random number generator with OpenSSL is provided.

An implementation of the CPU Jitter random number generator encapsulated into different OpenSSL Engines is provided. The relationship of the different engines to the OpenSSL default random number generator is depicted in Figure 27.

The following OpenSSL Engines are implemented:

jitterentropy-raw The `jitterentropy-raw` engine provides direct access to the CPU Jitter random number generator.

jitterentropy-drng The `jitterentropy-drng` engine generates random numbers out of the OpenSSL default deterministic random number generator. This DRNG is seeded with 16 bytes out of CPU Jitter random number generator every 1024 bytes. After 1,048,576 bytes, the DRNG is seeded and re-keyed, if applicable, with 48 bytes after a full reset of the DRNG. When the Note, the intention of this engine implementation is that it is registered as the default OpenSSL random number generator using `ENGINE_set_default_RAND(3)`.

jitterentropy-strong The `jitterentropy-strong` engine is very similar to `jitterentropy-drng` except that the reseeding values are 16 bytes and 1024 bytes, respectively. The goal of the reseeding is that always information theoretical entropy is present in the DRNG[6].

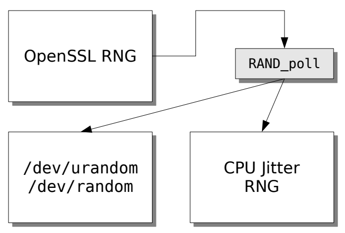


Figure 28: Linking OpenSSL with CPU Jitter RNG

The different makefiles compile the different engine shared library. The test case `tests_userspace/openssl/jitterentropy-eng-test.c` shows the proper working of the respective CPU Jitter random number generator OpenSSL Engines.

In addition, a patch independent from the OpenSSL Engine support is provided that modifies the `RAND_poll` API call to seed the OpenSSL deterministic random number generator. The `RAND_poll` first tries to obtain entropy from the CPU Jitter random number generator. If that fails, e.g. the initialization call fails due to missing high-resolution timer support, the standard call procedure to open `/dev/urandom` or `/dev/random` or the EGD is performed.

Figure 28 illustrates the operation.

The code is tested with the test application `tests_userspace/openssl/jent_test.c`. When using `strace` on this application, one can see that after patching OpenSSL, `/dev/urandom` is not opened and thus

[6]For the FIPS 140-2 ANSI X9.31 DRNG, this equals to one AES block. For the default SHA-1 based DRNG with a block size of 160 bits, the reseeding occurs a bit more frequent than necessary, though.

not used. That implies that the CPU Jitter random number generator code for seeding is invoked.

See `patches/README` for details on how to apply the code to OpenSSL.

E Shared Library And Stand-Alone Daemon

The CPU Jitter random number generator can be compiled as a stand-alone shared library using the `Makefile.shared` makefile. The shared library exports the interfaces outlined in `jitterentropy(3)`. After compilation, link with the shared library using the linker option `-ljitterentropy`.

To update the entropy in the `input_pool` behind the Linux `/dev/random` and `/dev/urandom` devices, the daemon `jitterentropy-rngd` is implemented. It polls on `/dev/random`. The kernel wakes up polling processes when the entropy counter falls below a threshold. In this case, the `jitterentropy-rngd` gathers 256 bytes of entropy and injects it into the `input_pool`. In addition, `/proc/sys/kernel/random/entropy_avail` is read in 5 second steps. If the value falls below 1024, `jitterentropy-rngd` gathers 256 bytes of entropy and injects it into the `input_pool`. The reason for polling `entropy_avail` is the fact that when random numbers are extracted from `/dev/urandom`, the poll on `/dev/random` is not triggered when the entropy estimator falls.

F Folding Loop Entropy Measurements

Measurements as explained in Section 5.1 for different CPUs are executed on a large number of tests on different CPUs with different operating systems were executed. The test results demonstrate that the CPU Jitter random number generator delivers high-quality entropy on:

- a large range of CPUs ranging from embedded systems of MIPS and ARM CPUs, covering desktop systems with AMD and Intel x86 32 bit and 64 bit CPUs up to server CPUs of Intel Itanium, Sparc, POWER and IBM System Z;
- a large range of operating systems: Linux, OpenBSD, FreeBSD, NetBSD, AIX, OpenIndiana (OpenSolaris), AIX, z/OS, and microkernel based operating systems (Genode with microkernels of NOVA, Fiasco.OC, Pistachio);
- a range of different compilers: GCC, Clang and the z/OS C compiler.

The listing of the test results is provided at the web site offering the source code as well.

G License

The implementation of the CPU Jitter random number generator, all support mechanisms, the test cases and the documentation are subject to the following license.

Policy-extendable LMK filter framework for embedded system

LMK: the state of the art and its enhancement

Kunhoon Baik
Samsung Electronics Co., Ltd.
Korea Advanced Institute of Science and Technology
`knhoon.baik@samsung.com, knhoon.baik@kaist.ac.kr`

Jongseok Kim
Korea Advanced Institute of Science and Technology
`paldad@kaist.ac.kr`

Daeyoung Kim
Korea Advanced Institute of Science and Technology
`kimd@kaist.ac.kr`

Abstract

Background application management by low memory killer (LMK) is one of the outstanding features of Linux-based platforms such as Android or Tizen. However, LMK has been debated in the Linux community because victim selection mechanism with a specific policy is not suitable for the Linux kernel and a flexible way to apply new policies has been required. Thus, several developers have tried implementing a userspace LMK like the ulmkd (userspace low memory killer daemon). However, not much work has been done regarding applying new polices.

In this paper, we present a policy-extendable LMK filter framework similar to the out-of-memory killer filter discussed at the 2013 LSF/MM Summit. The framework is integrated into the native LMK. When the LMK is triggered, each LMK filter module manages processes in the background like packet filters in the network stack. While the native LMK keeps background applications based on a specific policy, the framework can enhance background application management policy. We describe several benefits of the enhanced policies, including managing undesirable power-consuming background applications and memory-leaking background applications. We also present a new LMK filter module to improve the accuracy of victim selection. The module keeps the applications which could be used in the near future by predicting which applications are likely to be used next from the latest used application based on a Markov model.

We implemented the framework and the module on Galaxy S4 and Odroid-XU device using the Linux 3.4.5 kernel and acquired a preliminary result. The result shows that the number of application terminations was reduced by 14%. Although we implemented it in the kernel, it can be implemented as a userspace daemon by using ulmkd. We expect that the policy-extendable LMK filter framework and LMK filter will improve user experience.

1 Introduction

Modern S/W platforms for embedded devices support a background application management. The applications stacked in the background are alive until the operating system meets a specific condition such as memory pressure, if a user does not kill the applications intentionally. The background application management permits fast reactivation of the applications for later access [2], battery lifetime can be longer because energy consumed by applications re-loading can be reduced. [13]

However, applications cannot be stacked infinitely in the background because memory capacity is limited. Instead, the operating system needs to effectively manage the background applications in low memory situation. To handle such a situation, the operating system provides out-of-memory handler. Unfortunately, it causes significant performance degradation to user interactive

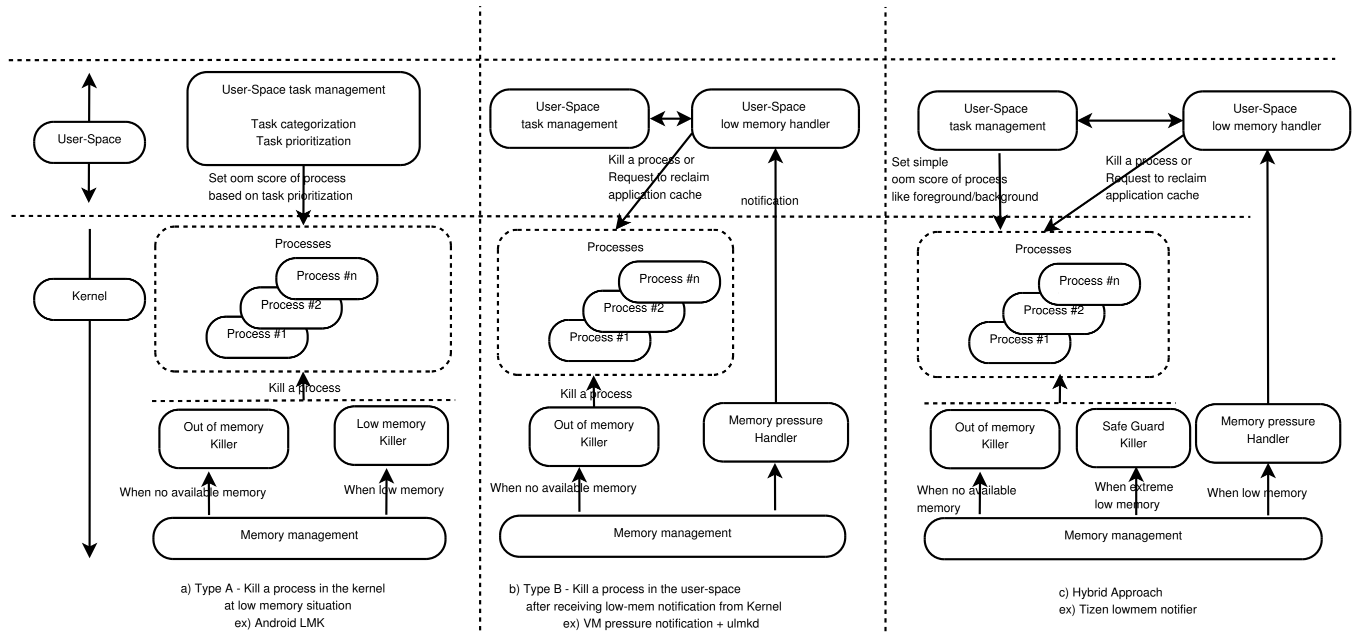

User-Space
Kernel
User-Space task management
Task categorization
Task prioritization
Set oom score of process
based on task prioritization
Processes
Process #n
Process #2
Process #1
Kill a process
Out of memory Killer
Low memory Killer
When no available memory
When low memory
Memory management
a) Type A - Kill a process in the kernel at low memory situation
ex) Android LMK
User-Space task management
User-Space low memory handler
Kill a process or Request to reclaim application cache
notification
Memory pressure Handler
b) Type B - Kill a process in the user-space after receiving low-mem notification from Kernel
ex) VM pressure notification + ulmkd
Set simple oom score of process like foreground/background
Safe Guard Killer
When extreme low memory
c) Hybrid Approach
ex) Tizen lowmem notifier

Figure 1: Architecture according to types of low memory killer

applications by excessive demand paging because the operating system triggers the OOM killer under desperately low memory conditions.

To avoid excessive demand paging, it was necessary for the operating system to trigger a low memory handler before falling into desperately low memory conditions. Several approaches have been introduced [10, 12, 6, 15, 16, 11]. The approaches can be categorized into two types from a structural perspective. Figure 1 shows the operation flow according to the types of low memory handler. The type-A approach, which is shown in Figure 1-(a), is to kill a process with the lowest priority in the kernel, according to the process priority configured in the user-space when a low memory handler is triggered. The type-B approach, which is shown in Figure 1-(b), is to kill a process after prioritizing processes in the user-space when a low memory handler is triggered. The type-A approach is hard to change the priority of a process in a low memory situation, and the type-B approach suffers from issues like latency until a process is killed after a low memory handler is triggered.

The Android low memory killer (LMK) is one of the type-A approaches, and it has been used for a long time in several embedded devices based on the Android platform. However, to apply a new victim selection policy, the user-space background application management must be modified, and it is impossible to re-prioritize the priority of processes in a low memory situation. Therefore, type-B approaches like userland low-memory killer daemon (ulmkd), have received attention again because the kernel layer provides simple notification functionality, and the approach can give the user-space an opportunity to dynamically change a victim selection policy. However, the user-space LMK is unlikely to handle a low memory situation in a timely way for a case of exhausted memory usage. Although related developers have made several attempts, the user-space LMK still has unresolved issues. [11] Thus, it is still too early to use the user-space LMK to dynamically apply a new victim selection policy.

While the type-B approach has been actively discussed and developed, applying new victim selection polices has not progressed, even though an advanced LMK victim selection policy would improve user experience and system performance. In the case of smartphones, most S/W platforms adopt the least recently used (LRU) victim selection algorithm to select a victim application in a low memory situation. However, the LRU victim selection algorithm sometimes selects applications to be used in the near future because the applications is likely to depend on the last-used application, hour of day, and location. [9] Likewise, the LRU victim selection algorithm does not preferentially select undesirable applications, such as memory-leaking applications, in a low memory situation. [13] Thus, if a system provides a mechanism to easily apply a new victim selection policy, it would improve the user experience and system performance.

In this paper, we propose an LMK filter framework to provide such a policy extension. To do that, we suggest a new architecture, modifying the type-A approach, because the type-B approach is still difficult to apply in a commercial device. We re-factored Android LMK to solve the limitation of the type-A approach. Based on the re-factored Android LMK, we created an LMK filter framework to provide real-time policy extension. The LMK filter framework provides the interfaces for managing policy modules with a policy extension, and the engine for applying the new policy of the policy modules. In addition, we present a task-prediction filter module to improve the accuracy of victim selection. With the results provided, we expect that the LMK filter framework and the module will improve user's experience.

The rest of this paper is organized as follows. Section 2 briefly describes previous approaches to memory overload management in the Linux kernel. Section 3 describes previous studies for applying a new victim selection policy. Section 4 provides the LMK filter framework and detailed implementation. Section 5 provides the task-prediction filter module to enhance the accuracy of victim selection. Section 6 shows a preliminary result for the suggested LMK filter framework and the module. The remaining sections offer discussion and conclusions.

2 Previous Approaches for Memory Overload Management in Linux

There have been several approaches to handling memory overload in swap-less devices. As described in Section 1, the approaches can be categorized into two types. The type-A approach is to give hints with `oom_score_adj` [1] to the kernel and kill a process in the kernel with the hints. The type-B approach is to kill a process in the user-space after receiving low memory notification from the kernel. In this section, we briefly describe these approaches.

[1]oom_score_adj is used as the adjustment of each process's attractiveness to the OOM killer. The variable is also used as hints in LMK

Category	Status
System/Persistent	Process is system or persistent process
Foreground	Process is in the foreground or related to the foreground application.
Visible	Process is visible or related to visible applications.
Perceptible	Process is not interacting with user but it can be perceptible to user.
Heavy	Process has cantSaveState flag in its manifest file.
Backup	Process has backup agent currently work on.
Service A/B	Process hosts service.
Home	Process is home application (like Android launcher)
Previous	Process was foreground application at previous.
Hidden	Process is in the background with no above condition.
Empty	Process has no activity and no service.

Table 1: Process Categorization of Android v4.3

2.1 Type-A Approach - Android Low Memory Killer

Android LMK is one of the type-A approaches. The Android platform gives hints to the Android LMK with `oom_score_adj`, and Android LMK selects a victim based on the given `oom_score_adj`. When the operating system triggers the Android LMK, Android LMK determines the minimum `oom_score_adj` of a process to be killed according to six-level memory thresholds and six-level `oom_score_adj` thresholds defined by the Android platform. Android LMK selects processes as victim candidates when they have an `oom_score_adj` higher than the minimum `oom_score_adj`, and it kills the process with the highest `oom_score_adj`. If there are several processes with the highest `oom_score_adj`, it selects the process with the largest memory as the final victim.

To give hints to the Android LMK, the Android platform categorizes processes according to the status of process components. Table 2.1 shows the process categorization of Android V4.3. Based on the categorization, the Android activity manager assigns the proper value to the `oom_score_adj` of each process. Thus, six thresholds of the Android LMK are closely related to the Android process categorization, because the Android LMK tries to kill an application in a specific categorization at a specific memory threshold.

Hidden or empty process categories in the low priority group must be prioritized. Android internally manages a processes list based on LRU according to the launched or resumed time of applications. For the hidden or empty process category, Android assigns process priority based on the LRU. That is, if a process has been more recently used, Android assigns a high priority value to that process. As a result, Android LMK is likely to kill the least recently used process in the hidden or empty process category.

Android LMK has been used for a long time in several embedded devices. However, it is not easy to apply a new victim selection policy because the activity manager of the Android or Android LMK must be modified.

2.2 Type-B Approach - Memory Pressure Notification

Several developers have tried to notify the user-space of memory pressure. The Nokia out-of-memory notifier is one of the early attempts. It attaches to the Linux security module (LSM). [10] Whenever the kernel checks that a process has enough memory to allocate a new virtual mapping, the kernel triggers the low memory handler. At that time, the module decides to send a notification to the user-space through the uevent if the number of available pages is lower than a user-defined threshold. The module sends a level1 notification or a level2 notification based on the user-defined thresholds. However, this method has been hard to use as a generic notification layer for a type-B approach because it only consider a user-space allocation request.

The mem_notify_patch is one of the generic memory pressure notifications. [6] The mem_notify patch sends a low memory notification to applications like the `SIGDANGER` signal of AIX. Internally, it was integrated into the page reclaim routine of the Linux kernel. It triggers a low memory notification when an anonymous page tries to move to the inactive list. If an application polls the "/dev/mem_notify" device node, the application can get the notification signal. The concept of the approach has led to other approaches like the Linux VM pressure notifications [12] and the mem-pressure control group [15]. Such approaches have improved the memory pressure notification.

In type-B approach, user-space programs have the responsibility of handling low memory notification with a policy because the memory pressure notification is the backend of the type-B approach. Thus, the type-B approach expects that the user-space programs release their cache immediately or kill themselves. However, the expectation is somewhat optimistic because all user-space programs may ignore the notification, or user-space programs may handle the notification belatedly. As a result, the system is likely to fall into out-of-memory in such situations. Thus, kernel/user-space mixed solutions, as shown Figure 1-(c), have been developed to improve the limitation of the type-B approach. The Tizen lowmem notifier is one of those hybrid approaches. [16] The Tizen lowmem notifier provides the low memory notification mechanism, and the safeguard killer kills an application based on the `oom_score_adj` given by a user-space daemon when the available memory of the system is very low. Thus, the safeguard killer prevents a system from falling into out-of-memory even when the user-space programs neglect to handle a low memory notification. However, the structure of the low memory handler is quite complicated.

2.3 Type-B Approach - Userland Low Memory Killer Daemon (ulmkd)

Ulmkd is one of frontend solutions of the type-B approach using generic memory pressure notification. The default notification backend of ulmkd is a low memory notification layer on top of cgroups. After receiving a memory notification from the kernel, ulmkd behaves the same way as the Android LMK driver by reading the `oom_score_adj` of each process from the proc file system. Thus, ulmkd needs to read the memory usage information of the system/process from the kernel. As a result, ulmkd can cause a lot of unnecessary system call in a low memory situation. In addition, the process page of ulmkd should not be reclaimed, to prevent unintended memory allocation by page remapping. Although the author of ulmkd tries to solve the issues by using approaches like locking of the process page, and using task-list management of specific user-space platform components, ulmkd still has issues to resolve. [11]

Although it requires lots of stability testing before applying to commercial devices, due to radical issues of user-space LMK, it is quite attractive because it make it easier to change the victim selection policy than the type-A approach. However, ulmkd does not provide a framework to apply the new victim selection policy dynamically or to apply multiple victim selection policies.

To the best of our knowledge, there are no existing solutions for extending the victim selection policy. A similar issue for the OOM killer was discussed at the 2013 LSF/MM summit and an idea to apply a new victim selection policy was suggested [3]. The idea is to create a framework similar to packet filters in the network stack. In this paper, we will present such a framework for dynamically extending victim selection policy.

3 Previous Studies for Enhancing Victim Selection Policy

Although mechanisms for handling the low memory situation is major topic in the industry, not much work has been done for enhancing victim selection policy. However, there are several benefits to enhancing the victim selection policy. In this section, we introduce such studies and benefits.

3.1 Considering Expected Delay

Yi-Fan Chung et al. utilized the concept of *expected delay penalty* to enhance victim selection [2]. *Expected delay penalty* was calculated by multiplying application launch probability by application launching time. If an application has a high *expected delay penalty*, the application is kept in the background. As a result, frequently used applications with long launch times are kept in the background instead of being killed.

3.2 Considering Undesirable Applications

Yi-Fan Chung et al. also utilized the concept of *expected power saving* to enhance victim selection. [2] After they calculated the power consumption of a background application, they calculated *expected power saving* by multiplying the application launch probability by the background power consumption of each background application. If an application has low *expected power saving*, it is kept in the background. Thus, less frequently used applications with high power consumption in the background are selected as victims.

Mingyuan Xia et al. studied how memory-leaking applications can easily cripple background application management with victim selection based on LRU. [13] They noted that normal applications lost the opportunity to be cached in the background when memory-leaking applications were kept in the background. They implemented a light weight memory-leak detector and they modified the Android process prioritization policy. If the detector detects a suspected memory-leaking application, the process is set to the lowest priority. Thus, Android LMK kills the suspected memory-leaking application.

3.3 Considering Personalized Factors

Tingxin Yan et al. predicted applications to be prelaunched by investigating application usage behaviour. [14] They investigated three application usage patterns: "follow-trigger", "location clustering", and "temporal burst". The application usage behaviours were used to predict which applications were likely to be used in the near futures, and the system assigned high priority to such applications. If applications with high priority did not exist in the background, the applications were prelaunched. Likewise, if applications with low priority existed in the background, the applications were selected as victims.

Predicting applications to be used in the near future can enhance victim selection policy. There have been several studies focused on predicting applications that will be used in the near future. Choonsung Shin et al. found that it was effective to predict applications from the last application used, and Cell ID and time of day. [9]. Xun Zou et al. showed a solution for predicting applications from the latest application used based on the Markov model. [17]

If a system provides a mechanism to easily apply the described victim selection policies, it would improve the user experience and system performance. In this paper, we present the LMK filter framework to apply such victim selection policies, and we show the system improvement by applying a new victim selection policy based on a prediction mechanism.

4 The Policy-extendable LMK Filter Framework

In this section, we present a policy-extendable LMK filter framework to extend new policies to an existing

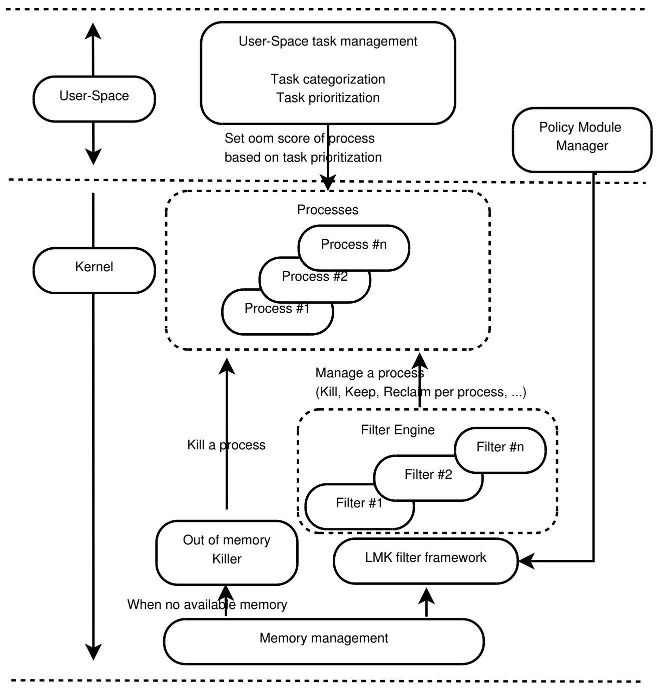


Figure 2: Architecture of LMK filter framework

LMK for a specific purpose or for general improvement. With the LMK filter framework, a module with a new victim selection policy can be applied at any time without modification. The purpose of the LMK filter framework is to provide following functionality.

- Adjust the victim selection policy of the LMK engine by adding/removing/changing policy modules;
- Support multiple policy modules effectively.

To adjust the policy of LMK in runtime, a policy is consisted of an independent device module. The policy of LMK is adjusted by managing the device module with a policy. To support multiple policy modules, the LMK filter framework implements a first-match resolution mechanism based on the order of policies. With the first-match resolution, the LMK filter framework minimizes a policy conflict and a policy redundancy by the installed several policies. In our implementation, we have termed a policy module and a first-match resolution engine to a filter module and a filter engine.

Figure 2 shows the architecture of the low memory handler with the LMK filter framework. The LMK filter framework and filters replace the LMK driver of the type-A approach. To create a policy-extendable LMK filter framework, we modified the Android LMK because the Android LMK has been used popularly. The type-B approach is more suitable for the LMK filter framework due to its flexibility. However, we chose the type-A approach because the type-B approach has still issues to solve. Although we implemented the LMK filter framework to the type-A approach, it is not difficult to apply to the type-B approach.

4.1 Re-factoring Android LMK

To create a generic LMK filter framework without changing the behaviour of the Android LMK, we analyzed the flow of the Android LMK. Figure 3-(a) shows the detail flow of Android LMK. Android LMK has a generic part for checking available memory, and for selecting an application as victim based on `oom_score_adj`. In addition, Android LMK has a specific part for filtering applications based on the six-level thresholds. Thus, we replaced the specific part with the filter engine routines of the LMK filter framework. Figure 3-(b) shows the re-factored Android LMK.

We replaced the "determine minimum `oom_score_adj` based on six-level thresholds" stage with a generic preprocessing stage for each filter module, and we replaced the "filter out processes based on minimum `oom_score_adj`" stage with a generic filtering stage for each filter module. Finally, we placed a generic postprocessing stage for each filter module after iterating processes. Thus, the sequence of the modified Android LMK is the following. The modified LMK checks the available memory including reclaimable memory, and checks the minimal memory threshold to fall routines for killing a process. After that, the LMK calls preprocessing routines for each filter module to prepare each filter module. After the filtering out processes performed by the filtering routines of each filter module, a process with the highest `oom_score_adj` or a process decided by a filter module is selected as victim, Finally, the LMK kills the victim after calling post-processing routines for each filter module. If the LMK filter framework does not have any filters, the LMK kills the process with the highest `oom_score_adj`. That is, the default behaviour of the LMK filter framework is to kill a process by generic decision based on `oom_score_adj`. We discuss the default behaviour further in Section 7

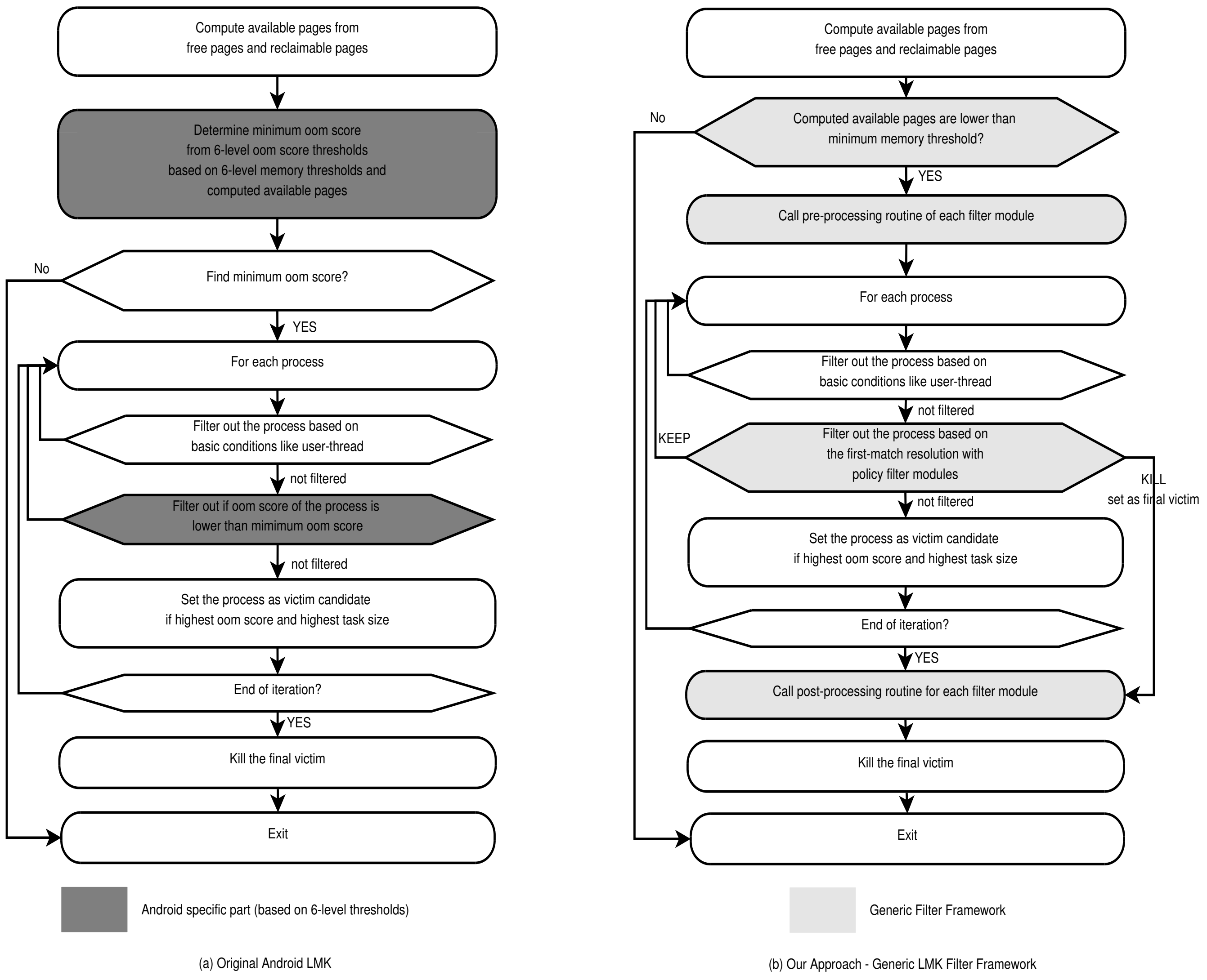

Compute available pages from free pages and reclaimable pages
Determine minimum oom score from 6-level oom score thresholds based on 6-level memory thresholds and computed available pages
Find minimum oom score?
No
YES
For each process
Filter out the process based on basic conditions like user-thread
not filtered
Filter out if oom score of the process is lower than mimimum oom score
not filtered
Set the process as victim candidate if highest oom score and highest task size
End of iteration?
YES
Kill the final victim
Exit
Android specific part (based on 6-level thresholds)
(a) Original Android LMK
Compute available pages from free pages and reclaimable pages
Computed available pages are lower than minimum memory threshold?
No
YES
Call pre-processing routine of each filter module
For each process
Filter out the process based on basic conditions like user-thread
not filtered
Filter out the process based on the first-match resolution with policy filter modules
KEEP
KILL set as final victim
not filtered
Set the process as victim candidate if highest oom score and highest task size
End of iteration?
YES
Call post-processing routine for each filter module
Kill the final victim
Exit
Generic Filter Framework
(b) Our Approach - Generic LMK Filter Framework

Figure 3: Comparison between native LMK and our approach

4.2 The LMK filter framework

The LMK filter framework consists of two parts: the filter chain manager and the filter engine. The filter chain manager manages the list of filter modules and the life-cycle of filter modules. The manager supports to add a new filter module to the LMK filter engine, and to remove an existing filter module from the LMK filter engine. In addition, the manager supports to change the order of filter modules for the first-match resolution.

The filter engine is the core of the LMK filter framework. The filter engine operates the first-match resolution in the low memory situation and exposes filtering interfaces for filter modules. Figure 4 shows the first-match resolution flow for a process. If a filter module decides that a process should be preserved in the background in the filtering stage, that process is not killed and the traversing filter chain for the process is terminated. Likewise, if a filter module decides that a process should be killed, the filter engine sets the process as a victim and the traversing filter chain of the process is terminated. If a filter module does not make any decision for a process, the filter engine calls the filtering routine of the next filter module in the filter chain. This means that filter modules have a priority based on their position in the filter chain.

The filter engine provides an interface for a filter module. With the interface, each filter module is required to register three routines: a pre-processing routine, post-processing routine, and filtering routine. As shown in Figure 3-(b), the pre-processing and post-process rou-

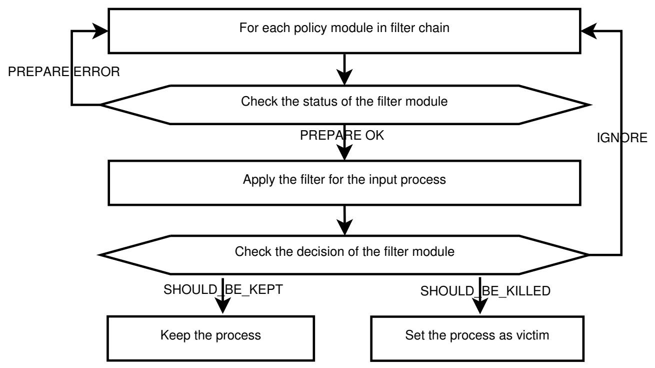

For each policy module in filter chain
PREPARE ERROR
Check the status of the filter module
PREPARE OK
IGNORE
Apply the filter for the input process
Check the decision of the filter module
SHOULD_BE_KEPT
SHOULD_BE_KILLED
Keep the process
Set the process as victim

Figure 4: The first-match resolution of filter engine

tines are called once in a low memory handling flow, and the filtering routine is called for every process, whenever the filter engine iterates processes.

In a pre-processing routine, a filter module must run preparation to filter a process like setting a filter-owned threshold. A module should return PREPARE_DONE or PREPARE_ERROR in the routine. If a module returns PREPARE_ERROR for any reason, the filter engine excludes the filter module in the low memory handling flow. In a filtering routine, a filter module can filter a process based on the filter module's specific victim selection policy. In a filtering routine, a filter module can return one of three decisions for each process: IGNORE, SHOULD_BE_VICTIM, SHOULD_BE_KEPT. If a filter module returns SHOULD_BE_VICTIM for a process, the process is set as a victim. If a filter module returns SHOULD_BE_KEPT, the process is kept in the background. When a filter module does not make any decision for a process, the filter module can return IGNORE.

4.3 Android Specific Filter Module

To execute the same operation with a native Android LMK, we implemented a filter module for the Android. The module implements the a specific part of Android LMK.

In the pre-processing routine, the module finds the minimum oom_score_adj from the six-level oom_score_adj thresholds defined by a user-space platform after comparing the six-level memory thresholds defined by a user-space platform with the calculated available memory pages. If the module finds the minimum oom_score_adj, the module returns a PREPARE_DONE value. In the filtering routine, the module compares the oom_score_adj of a process to the minimum oom_score_adj decided in the preparation stage. If the oom_score_adj of a process is higher than the minimum oom_score_adj, the module returns IGNORE. Otherwise, the module returns SHOULD_BE_KEPT. Thus, if the oom_score_adj of processes are lower than the minimum oom_score_adj, the processes will be kept in the background. If there are other filters, other filters modules will decide the termination of the ignored processes.

5 The Task-Prediction Filter Module

To show a policy-extension with the LMK filter framework, we implemented a new policy module. The victim selection in Android is decided by application categorization and LRU. Therefore, the Android LMK may select an application to be re-used in the near future as a victim because recently used processes may not be reused in the near future. The accuracy of victim selection can be improved by carefully predicting applications what will be reused in the near future. In particular, the last application provides hints to predict an application reused in the near future [9]. To extend the study, we considered the last N-applications. In addition, we considered the memory used by a process to give a penalty for a process using large memory. Thus, we suggest a filter module to keep processes to be reused in the near future based on the last N-application and process memory usage.

The main functionality of the filter module consists of two parts. The first is to maintain an application transition probability matrix and an application usage history, and the other is the filter-preparation function and the filtering function of the LMK filter module provided for keeping the predicted applications in a low memory situation.

5.1 Training prediction matrix

To predict which application would be used next from last N-applications, the transition probability between applications was observed during a training phase. The trained transition probability matrix was used as a Markov chain matrix to predict applications which could be used in the next step. To determine the transition probability between applications, we tracked the foreground application of the Android. To track the foreground application, we modified the write operation of `proc_oom_adjust_operations` in the Linux kernel. Whenever the `oom_score_adj` of a process is changed, the `oom_score_adj` hook function of our filter module is called. The hook function logs the foreground application, which has the foreground `oom_score_adj` of the Android platform. When the hook function recognizes a foreground application, the module inserts the information of the application into a queue which has finite-length to maintain the application usages history, and the module updates the information of a transition probability matrix by relationship.

5.2 Implementation of the filter module

To predict applications to be reused in the near future, we utilized the Markov chain models for link prediction of Ramesh R. Sarukkai.[7] From the model, a probability of the next transition from n-past information can be acquired. Thus, the probability of next application's transition from last N-applications is acquired with the model.

Let M represent the trained application transition probability matrix, and let S(t) represent the state of a foreground application at time t. The following formula derives the transition probability using the Markov chain models for link prediction of Ramesh R. Sarukkai.

$$s(t+1) = \alpha_1 s(t)M + \alpha_2 s(t-1)M^2 + \alpha_3 s(t-2)M^3 + \ldots$$

In the pre-processing routine, the filter module prepares the transition probability matrix for the formula. In the filtering routine, we generate the final transition probability from the formula.

To filter a process, we considered an additional factor – the memory usage of a process. Although the transition probability of a process is higher than others, if the memory usage of the process is higher than others, it is possible that keeping the process will cause a decrease in the total number of applications kept in the background. That is, when the prediction is incorrect, keeping a process with large memory is likely to produce side-effects. Thus, to reduce such side-effect, the memory usage of a process was also considered with the transition probability.

Based on the transition probability derived from the formula and the memory usage of a process, the filter module computes a transition score. If an application has a high transition probability and low memory usage, the application has a high transition score. Otherwise, if an application has a low transition probability and high memory usage, the application has a low transition score.

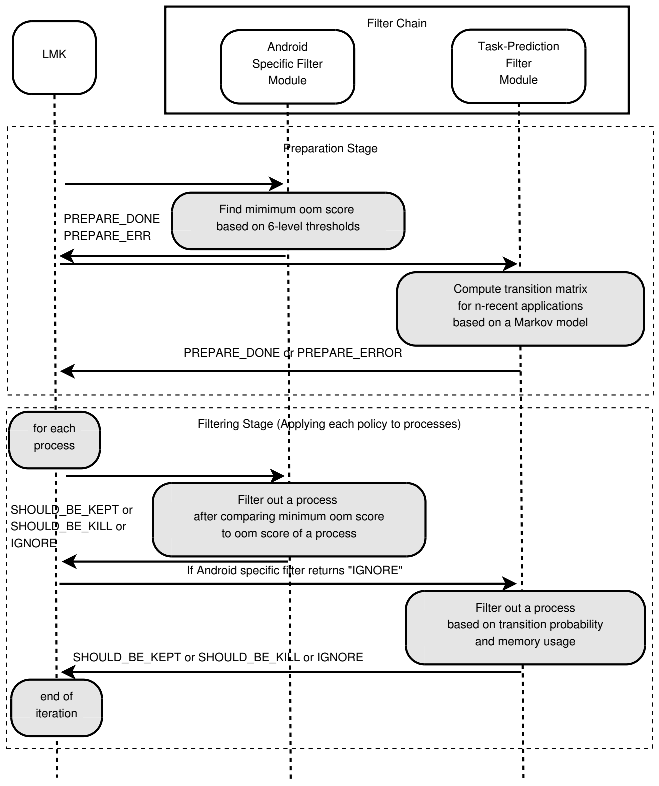

LMK
Filter Chain
Android Specific Filter Module
Task-Prediction Filter Module
Preparation Stage
Find mimimum oom score based on 6-level thresholds
PREPARE_DONE
PREPARE_ERR
Compute transition matrix for n-recent applications based on a Markov model
PREPARE_DONE or PREPARE_ERROR
Filtering Stage (Applying each policy to processes)
for each process
Filter out a process after comparing minimum oom score to oom score of a process
SHOULD_BE_KEPT or
SHOULD_BE_KILL or
IGNORE
If Android specific filter returns "IGNORE"
Filter out a process based on transition probability and memory usage
SHOULD_BE_KEPT or SHOULD_BE_KILL or IGNORE
end of iteration

Figure 5: LMK filter flows with the task-prediction filter module

As a result, the filtering function of the filter module keeps applications with a high transition probability and low memory usage, after computing the transition score. If a process has a high transition score, the module returns `SHOULD_BE_KEPT` for the process. Otherwise, the module returns `IGNORE`. Thus, the filter module tries to keep applications that will be reused in the near future from recent applications with minimal side effects.

The task-prediction filter module is inserted as the second filter module in Android device. Thus, the filter module enhances the LRU-based victim selection policy of Android without changing a victim selection policy by application categorization. Figure 5 shows the sequence of the LMK decision after inserting the task-prediction filter module with an Android specific filter module. The filter module decides the life of a process when the Android specific filter module does not make a decision about the life of the process.

6 Evaluation

In this section, we evaluate the effectiveness of the LMK filter framework and the task-prediction filter module. We implemented the LMK filter framework and the filter module on a Galaxy S4 and Odroid-XU using the Linux 3.4.5 kernel.

Android platform supports process-limit feature to limit the number of background applications. Based on the number of process limitation, Android platform kills an application in background application cache. Thus, if a device has enough memory, the LMK is rarely triggered because Android platform is likely to manage background applications based on process limit before falling into low memory situation. Unfortunately, two devices in our experiments equips enough memory for stacking the default number of process-limit. Thus, the LMK is rarely happen in the devices. Thus, to evaluate our algorithm, we adjusted the amount of memory of the two devices instead of increasing the process limit of Android. We discuss the process limit further in Section 7.

To evaluate our scheme, we created an evaluation framework based on real usage data in a smartphone. With the evaluation framework, we show that the number of application terminations is reduced. In a specific application usage sequence, the decrease of the number of application terminations means the increase of the hit rate for the background application reuse. In addition, it means the decrease of the data loss probability by the LMK forced termination.

6.1 Evaluation Framework

We created an evaluation framework to run applications automatically according to a given input sequence of applications. The framework launches an application in order from the given input applications sequence. To launch an application, the framework executes an activity manager tool through the shell command of android debug bridge (adb). The framework reads the input sequence of application and launches the next application every 10 seconds. At the same time it is performing the application launch, the evaluation framework monitors the number of application terminations provided by the LMK filter framework.

We referenced the usage sequence of applications collected from the Rice LiveLab user study to evaluate our scheme [8, 4]. We generated the input sequence by finding corresponding Android applications in Google Play after extracting a part of the original sequence. Table-2 shows information about the input sequences of each users.

User	Length of input seq.	Usage duration
A00	162	4 days 10 hours
A01	139	1 day
A03	233	2 days 21 hours
A07	154	6 days 11 hours
A10	218	8 days 8 hours
A11	179	6 days 19 hours
A12	159	3 days 9 hours

Table 2: Extracted input sequences

6.2 Estimation

We evaluated our frameworks by using 3-fold cross validation with each user's application usage data. One third of the total sequence was used for training set. We also conducted online training when a test set is tested. The task-prediction filter module predicted next applications from 1-recent applications history to 4-recent applications history. Each user's application usage data were tested five times, and we took the average from the results. Figure 6 shows the number of application terminations in both native LMK and the LMK filter framework with the task-prediction filter module when the filter module predicts the next application from 4-recent applications history. The result was normalized to the number of application terminations in native LMK.

The LMK filter framework with the task-prediction filter module was improved by average 14% compared with the native LMK. In addition, our proposed scheme gave better results for most users. In case of specific user a10, the number of application terminations was improved by 26%. Although we did not experiment with all real usage data, the preliminary result proves that an enhanced application replacement algorithm can be created easily by the LMK filter framework without modifying the entire process prioritization. In addition, in case of specific user a11, the number of application terminations was increased. It shows that a specific policy is not easy to fit to all users because the most appropriate policy for each user differ. The LMK filter framework can solve such problems by applying other policy modules for each user.

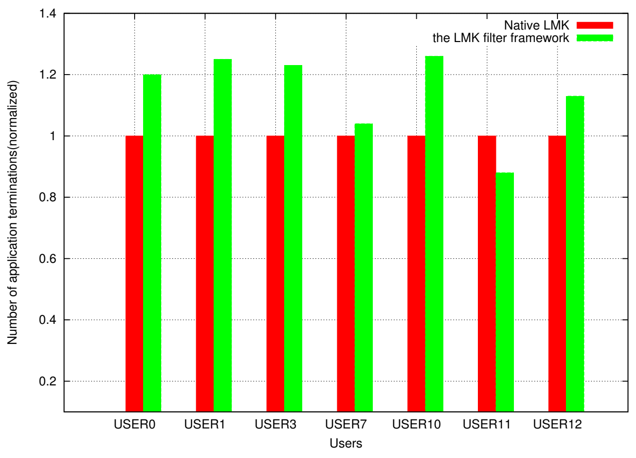


Figure 6: Number of application termination in both native LMK and LMK filter framework with the task-prediction filter module

Routine	N-recent applications	Overheads
Pre-Processing	1	31549 ns
	2	74020 ns
	3	109090 ns
	4	146699 ns
Filtering	1	3371 ns
	2	3522 ns
	3	3632 ns
	4	3411 ns

Table 3: Average overheads according to n-recent applications

6.3 Overheads

We observed the overheads of the LMK filter frameworks and filter modules. The overheads was measured in the pre-processing stage and the filtering stage. Table 6.3 shows the overheads. Most overheads was caused by the task-prediction filter module. In the pre-processing stage, the task-prediction filter module multiplies the transition matrix by n times according to n-recent applications history. Thus, the overheads were increased according to n. In the filtering stage, the filter module computes transition scores. The table shows computation the overheads of computing transition score about a process. Thus, the total overheads can be obtained by multiplying the number of iterated processes in the LMK filter framework. In our experiments, it was not significant overheads because the average number of iterated processes was about 10.

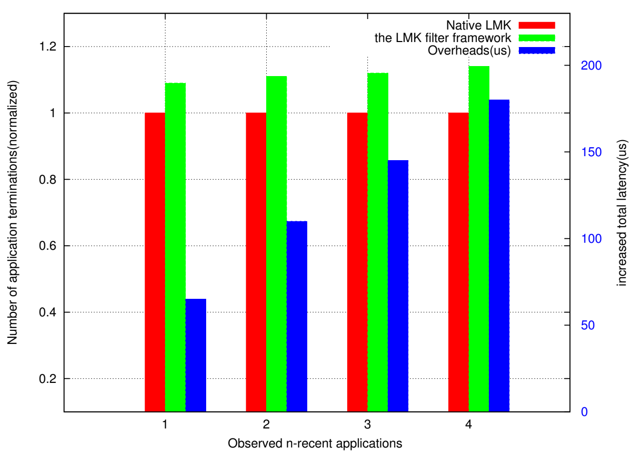


Figure 7: Average number of application terminations and overheads according to n-recent applications

6.4 Accuracy of the Task-Prediction Filter Module

The task-prediction filter module enhances the accuracy of victim selection by predicting the next application from the n-recent applications history. If many n-recent applications are used, application prediction is more accurate. However, the computation overhead is also increased significantly by matrix multiplications. Figure 7 shows the number of application terminations and overheads according to n-recent applications history. As the number of recent applications used for prediction increased, the number of application terminations was slightly reduced. However, the performance overhead is also increased by matrix multiplication. Thus, the n value should be applied properly according to the degree of urgency when handling a low memory situation.

7 Discussion

7.1 The default behaviour of the LMK Filter Framework

The default behavior of LMK filter framework is to kill a process with the highest `oom_score_adj`, and to kill a process with the largest memory when there are several processes with the highest `oom_score_adj`. However, such behavior may cause a reduction in the total number of applications kept in the background. If many applications with large memory are kept in the background, the total number of applications kept in the background is reduced because the memory capacity of the device is limited. Thus, the OOM killer of the kernel decides the badness of a process based on the percentage of process memory in total memory. The `oom_score_adj` of a process is used to adjust it. As a result, the behavior of the OOM killer is likely to keep more applications in the background.

For embedded devices, we believe that the `oom_score_adj` of applications given by a specific rules of user-space, such as application categorization of Android, is more important than the percentage of process memory in total memory because such hints include user experience related factors, such as visibility to user. However, this should be decided carefully after further experiments in several embedded devices with several users. To do that, the default behavior of the LMK filter framework also should be changeable. The works are left as our future work.

7.2 The first-match resolution mechanism

The LMK filter framework implements a first-match resolution mechanism. The mechanism is effective to minimize policy conflicts and to reduce redundant operations by policy redundancy. However, to integrate several policies, the order of policy modules should be decided carefully. For example, suppose that there are three policy modules: a memory-leak based policy module, the task-prediction module, the Android specific module. The memory-leak based policy module selects a memory-leak suspected application as victim, as described Section-3. If the order of the policy module is "the Android specific module – the task-prediction module – the memory-leak based policy module", a memory-leak suspected application can be kept in the background by previous two policy modules. Thus, the order of policy module should considered carefully for desirable integration of several policy modules.

7.3 Other Benefits of the LMK Filter Framework

The specific victim selection policies described in Section-3 were applied by individually modifying the process prioritization routine of the Android platform. However, the policies can be applied by using the LMK filter framework without platform modification. In addition, the policies can be activated at the same time by the LMK filter framework.

Instead of applying a new victim selection policy, a new memory management policy for processes can be applied with the LMK filter framework. For example, per-process page reclamation can be implemented as a filter module. A filter module can reclaim a process's page in the filtering routine after checking a process's reclaimable memory. After per-process reclamation, if the filter module returns `SHOULD_BE_KEPT` for the process, the process will be kept in the background. Without such a mechanism, although per-process page reclamation is provided to the user-space [5], the process is likely to be killed by LMK in a low memory situation.

7.4 User-space LMK filter framework

Implementing policies in the user-space might allow polices to be applied gracefully because user-space's information can be utilized easily. Meanwhile, the user-space program is unlikely to acquire OS-dependent information like the page-mapping information of the process. Above all, the user-space low memory killer is hard to handle quickly for urgently handling a low memory situation.

Thus, there are advantages and disadvantages with the in-kernel LMK filter framework. We expect that the in-kernel LMK filter framework will be able to apply new polices gracefully if the operating system manages context information, such as the context-aware OS service, to enhance the operating system's behaviour [1]. However, we also believe that the LMK filter framework can be easily applied to the user-space LMK, and the user-space LMK filter framework can apply various victim selection policies dependent on specific applications.

7.5 Victim Selection by Process Limit

A S/W platform for embedded device, such as Android, manages background applications based on the number of process limit. If the number of background processes exceeds a pre-defined number of process limitation, the S/W platform kills an application. Unfortunately, in the type-A approach like Android, the victim selection by process limit is executed in the user-space, and the victim selection by LMK is executed in the kernel. Thus, they try to synchronize the victim selection policy by hints, such as `oom_score_adj` of a process. In our implementation, we did not consider the synchronization. However, the problem can be easily solved by querying victim selection to LMK filter framework.

8 Conclusion

We presented a policy-extendable LMK filter framework and a filter module to enhance LMK victim selection, and showed the effectiveness of the customized LMK and several benefits provided by the LMK filter framework. Although we implemented it in the kernel, it can be implemented as a user-space daemon. We expect that this policy-extendable LMK filter framework and LMK filter will improve user experience.

Acknowledgments

This research was supported by the International Research & Development Program of the National Research Foundation (NRF) funded by the Ministry of Science, ICT & Future Planning (Grant number: 2013064540).

References

[1] David Chu, Aman Kansal, Jie Liu, and Feng Zhao. Mobile apps: It's time to move up to condos. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 16–16, Berkeley, CA, USA, 2011. USENIX Association.

[2] Yi-Fan Chung, Yin-Tsung Lo, and Chung-Ta King. Enhancing user experiences by exploiting energy and launch delay trade-off of mobile multimedia applications. *ACM Trans. Embed. Comput. Syst.*, 12(1s):37:1–37:19, March 2013.

[3] Jonathan Corbet. Lsfmm: Improving the out-of-memory killer. http://lwn.net/Articles/146861/, 2013.

[4] Rice Efficient Computing Group. Livelab: Measuring wireless networks and smartphone users in the field. http://livelab.recg.rice.edu/traces.html/.

[5] Minchan Kim. mm: Per process reclaim. http://lwn.net/Articles/544319/, 2013.

[6] KOSAKI Motohiro. mem_notify v6. http://lwn.net/Articles/268732/, 2008.

[7] Ramesh R. Sarukkai. Link prediction and path analysis using markov chains. In *Proceedings of the 9th International World Wide Web Conference on Computer Networks : The International Journal of Computer and Telecommunications Netowrking*, pages 377–386, Amsterdam, The Netherlands, The Netherlands, 2000. North-Holland Publishing Co.

[8] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. Livelab: Measuring wireless networks and smartphone users in the field. *SIGMETRICS Perform. Eval. Rev.*, 38(3):15–20, January 2011.

[9] Choonsung Shin, Jin-Hyuk Hong, and Anind K. Dey. Understanding and prediction of mobile application usage for smart phones. In *Proceedings of the 2012 ACM Conference on Ubiquitous Computing*, UbiComp '12, pages 173–182, New York, NY, USA, 2012. ACM.

[10] William McBride Aderson Traynor. Nokia out of memory notifier module. http://elinux.org/Accurate_Memory_Measurement#Nokia_out-of-memory_notifier_module, 2006.

[11] Anton Vorontsov. Userspace low memory killer daemon. https://lwn.net/Articles/511731/, 2012.

[12] Anton Vorontsov. vmpressure_fd: Linux vm pressure notifications. http://lwn.net/Articles/524299/, 2012.

[13] Mingyuan Xia, Wenbo He, Xue Liu, and Jie Liu. Why application errors drain battery easily?: A study of memory leaks in smartphone apps. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 2:1–2:5, New York, NY, USA, 2013. ACM.

[14] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 113–126, New York, NY, USA, 2012. ACM.

[15] Bartlomiej Zolnierkiewicz. The mempressure control group proposal. http://lwn.net/Articles/531077/, 2008.

[16] Bartlomiej Zolnierkiewicz. Efficient memory management on mobile devices. In *LinuxCon*, 2013.

[17] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. Prophet: What app you wish to use next. In *Proceedings of the 2013 ACM Conference on Pervasive and Ubiquitous Computing Adjunct Publication*, UbiComp '13 Adjunct, pages 167–170, New York, NY, USA, 2013. ACM.

Scalable Tools for Non-Intrusive Performance Debugging of Parallel Linux Workloads

Robert Schöne* Joseph Schuchart* Thomas Ilsche* Daniel Hackenberg*

*ZIH, Technische Universität Dresden
{robert.schoene|joseph.schuchart|thomas.ilsche|daniel.hackenberg}@tu-dresden.de

Abstract

There are a variety of tools to measure the performance of Linux systems and the applications running on them. However, the resulting performance data is often presented in plain text format or only with a very basic user interface. For large systems with many cores and concurrent threads, it is increasingly difficult to present the data in a clear way for analysis. Moreover, certain performance analysis and debugging tasks require the use of a high-resolution time-line based approach, again entailing data visualization challenges. Tools in the area of High Performance Computing (HPC) have long been able to scale to hundreds or thousands of parallel threads and are able to help find performance anomalies. We therefore present a solution to gather performance data using Linux performance monitoring interfaces. A combination of sampling and careful instrumentation allows us to obtain detailed performance traces with manageable overhead. We then convert the resulting output to the Open Trace Format (OTF) to bridge the gap between the recording infrastructure and HPC analysis tools. We explore ways to visualize the data by using the graphical tool Vampir. The combination of established Linux and HPC tools allows us to create an interface for easy navigation through time-ordered performance data grouped by thread or CPU and to help users find opportunities for performance optimizations.

1 Introduction and Motivation

GNU/Linux has become one of the most widely used operating systems, ranging from mobile devices, to laptop, desktop, and server systems to large high-performance computing (HPC) installations. Performance is a crucial topic on all these platforms, e.g., for extending battery life in mobile devices or to ensure maximum ROI of servers in production environments. However, performance tuning is still a complex task that often requires specialized tools to gain insight into the behavior of applications. Today there are only a small number of tools available to developers for understanding the run-time performance characteristics of their code, both on the kernel and the user land side. Moreover, the increasing parallelism of modern multi- and many-core processors creates an additional challenge since scalability is usually not a major focus of standard performance analysis tools. In contrast, scalability of applications and performance analysis tools have long been topics in the High Performance Computing (HPC) community. Nowadays, 96.4 % of the 500 fastest HPC installations run a Linux OS, as compared to 39.6 % in 2003[1]. Thus, the HPC community could benefit from better integration of Linux specific performance monitoring interfaces in their tools, as these are currently targeting parallel programs and rely on instrumenting calls to parallelization libraries such as the Message Passing Interface (MPI) and OpenMP. On the other hand, the Linux community could benefit from more scalable tools. We are therefore convinced that the topic of performance analysis should be mutually solved by bringing together the expertise of both communities.

In this paper, we present an approach towards scalable performance analysis for Linux using the perf infrastructure, which has been introduced with Linux 2.6.31 [8] and has undergone intensive development since then. This infrastructure allows users to access hardware performance counters, kernel-specific events, and information about the state of running applications. Additionally, we present a new visualization method for ftrace-based kernel instrumentation.

[1]Based on November 2003 and November 2013 statistics on http://top500.org

Measurement Type	Kernel Interface	Common Userspace Tools and Libraries
Instrumentation	ptrace	gdb, strace, ltrace
	ftrace	trace-cmd, kernelshark, ktap
	kernel tracepoints	LTTng, SystemTap, ktap, perf userspace tools
	dynamic probes	SystemTap, ktap, perf userspace tools
Sampling	perf events	perf userspace tools, PAPI
	OProfile (kernel module)	OProfile daemon and tools

Table 1: Common Linux Performance Analysis Interfaces and Tools

The remainder of this paper is structured as follows: Section 2 presents an overview of existing Linux performance analysis tools. Section 3 outlines the process of acquiring and processing performance data from the perf and ftrace infrastructures followed by the presentation of different use-cases in Section 4.

2 Linux Performance Monitoring Interfaces and Established Tools

Several interfaces are available in the Linux kernel to enable the monitoring of processes and the kernel itself. Based on these interfaces, well-established userspace tools and libraries are available to developers for various monitoring tasks (see Table 1). The *ptrace* [15] interface can be used to attach to processes but is not suitable for gaining information about the performance impact of kernel functions. *ftrace* [7] is a built-in instrumentation feature of the Linux kernel that enables kernel function tracing. It uses the `-pg` option of `gcc` to call a special function from every function in a kernel call. This special function usually executes NOPs. An API, which is located in the Debugfs, can be used to replace the NOPs with a tracing function. `trace-cmd` [25] is a command line tool that provides comfortable access to the ftrace functionality. *KernelShark* [26] is a GUI for `trace-cmd`, which is able to display trace information about calls within the Linux kernel based on ftrace events. This allows users to understand the system behavior, e.g., which processes trigger kernel functions and how tasks are scheduled. However, the KernelShark GUI is not scalable to large numbers of CPU cores and does not provide integration of sampling data, e.g., to present context information about application call-paths. Nevertheless, support for ftrace is currently being merged into the perf userspace tools [16]. Kernel *tracepoints* [3] are instrumentation points in different kernel modules that provide event-specific information, e.g., which process is scheduled to which CPU for a scheduling event or what hints have been used when allocating pages. *kprobes* are dynamic tracepoints that can be added to the kernel at run-time [12] by using the `perf probe` command. Such probes can also be inserted in userspace programs and libraries (*uprobes*). The *perf_event* infrastructure can handle kprobes and uprobes as well as tracepoint events. This allows the perf userspace tools to record the occurrences of these events and to integrate them into traces. The *Linux Trace Toolkit next generation* (*LTTng*) [10,11] is a tracing tool that allows users to measure and analyze user space and kernel space and is scalable to large core counts. It writes traces in the *Common Trace Format* which is supported by several analysis tools. However, these tools do not scale well to traces with large event counts. *SystemTap* [28] provides a scripting interface to access and react on kernel probes and ftrace points. Even though it is possible to write a generic (kernel) tracing tool with `stap` scripts, it is not intended for such a purpose. *ktap* is similar to SystemTap with the focus on kernel tracing. It supports tracepoints, dynamic probes, ftrace, and others.

In addition to the instrumentation infrastructure support in the kernel, measurement points can also be triggered by sampling. The *perf_event* infrastructure provides access to hardware-based sampling that is implemented on x86 processors with performance monitoring units (PMUs) that trigger APIC interrupts [5,14]. On such an interrupt, the call-graph can be captured and written to a trace, which is usually done with the `perf record` command-line tool but can also be achieved with low-level access to a memory-mapped buffer that is shared with the kernel. In a post-mortem step, tools like `perf script` and `perf report` use debugging symbols to map the resulting events in a trace file recorded by `perf record` to function names. *PAPI* [6,23] is the de facto standard library for reading performance counter information and is supported by most HPC tools. On current

Linux systems, PAPI uses the perf_event interface via the libpfm4 library. In addition to performance counter access, PAPI is also able to use this interface for sampling purposes. The OProfile kernel module [19] is updated regularly to support new processor architectures. It provides access to hardware PMUs that can be used for sampling, e.g., by the OProfile daemon [20].

However, none of the Linux performance analysis tools are capable of processing very large amounts of trace data, and none feature scalable visualization interfaces. Scalable HPC performance analysis tools such as Vampir [24], Score-P [18], HPCToolkit [1], and TAU [27], on the other hand, usually lack the close integration with the Linux kernel's performance and debugging interfaces.

3 Performance Data Acquisition and Conversion

In this section, we discuss our approach to obtaining performance data using standard tools and interfaces and how we further process the data to make it available to scalable analysis tools.

3.1 Data Acquisition with perf and ftrace

We use `perf record` to capture hardware-counter-based samples and selected tracepoints. In more detail, we use the following event sources:

`cpu-cycles`
This event is used as a sampling timer. Unlike typical alerts or timers, the cpu-cycles counter does not increase when the CPU is idle. Information about idling CPUs or tasks is not crucial for performance analysis and a lower interrupt rate in such scenarios minimizes the sampling overhead.

`sched_process_{fork|exec|exit}`
These tracepoint events are used to track the creation and termination of processes.

`sched_switch`
This tracepoint event is used to track processes on the CPUs. It provides knowledge about when which task was scheduled onto which CPU. The state of the task that is scheduled away is associated to the event in order to distinguish between voluntary sleep (state S), un-interruptible sleep (state D, usually I/O), or preemption (state R)[2].

[2]cf. `man top`

`instructions|cache-misses|...`
Other performance counters can be included in the timeline to get a better understanding of the efficiency of the running code. For example, the *instruction* counter allows us to determine the instruction per cycle (IPC) value for tasks and CPUs and adding *cache-misses* provides insights into the memory usage.

As an alternative to sampling, instrumentation can provide fine-grained information regarding the order and context of function calls. For debugging purposes, sampling is not a viable option. Thus, we use the kernel tracing infrastructure *ftrace* to analyze kernel internal behavior. One alternative would be a combination of *trace-cmd* and *KernelShark*. However, KernelShark is limited in terms of scalability and visualization of important information. Instead of `trace-cmd` we use a shell script to start and stop the kernel monitoring infrastructure ftrace. The script allows us to specify the size of the internal buffer for events and filters that can be passed to ftrace in the respective `debug fs` files. To create the trace, we enable the *function_graph* tracer and set the options to display overruns, the CPU, the process, the duration, and the absolute time. The script then starts the recording by enabling ftrace and stops it when the recording time expires.

3.2 Conversion to Scalable Data Formats

The `perf record` tool and its underlying file format are designed to induce only minimal overhead during measurement. It therefore simply dumps data from the kernel buffer directly into a single file without any distinction between process IDs or CPUs. This file can be used in a follow-up step to create a profile based on the recorded trace data using `perf report`. External tools can be used with `perf script` to analyze the trace data. However, the simple file structure resulting from the low-overhead recording process has negative side effects on the scalability of the data format. Parallel parsing of a single file is impeded by the variable length of single trace entries and the mixture of management information (e.g., task creation and termination) with performance event information from sampling.

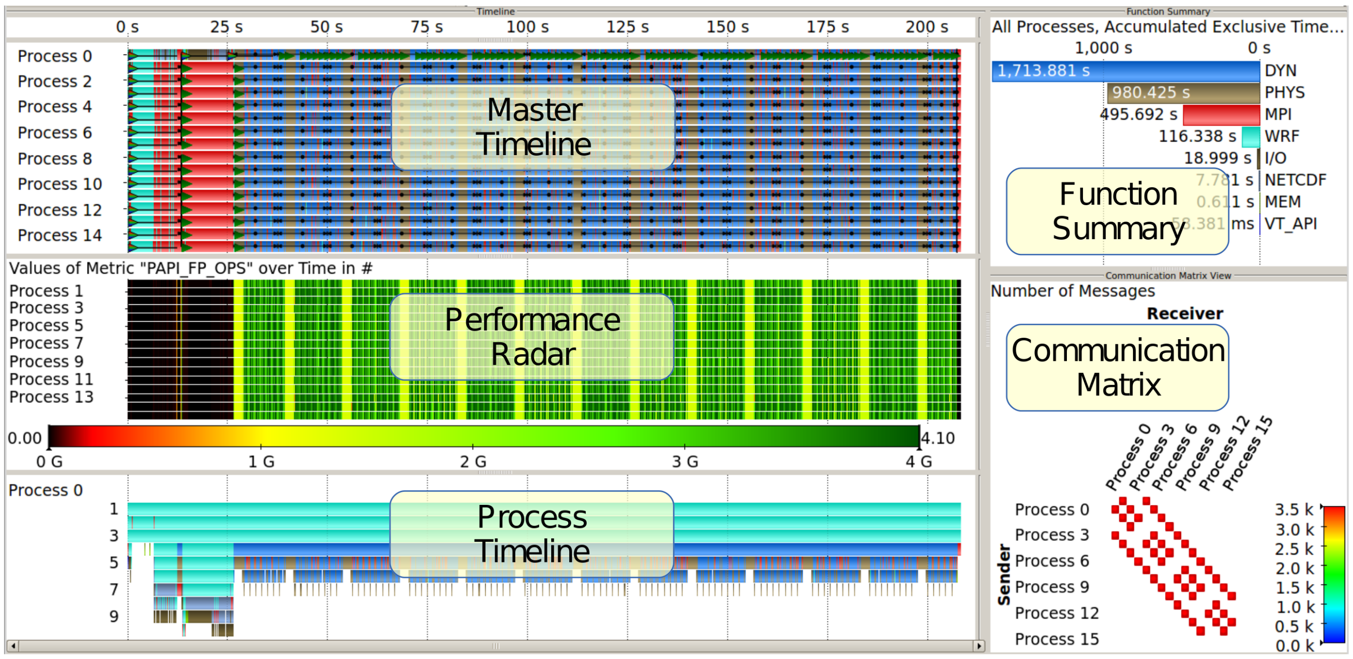


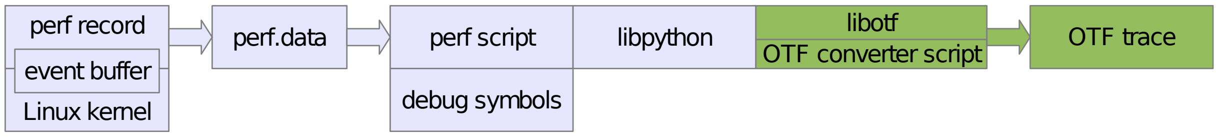
Figure 1: Vampir visualization of a trace of the HPC application WRF, including the *Master Timeline* showing the parallel process activity (different function calls in different colors, MPI messages as black lines aggregated in bursts), the *Performance Radar* depicting performance metrics such as hardware counter readings, and the *Process Timeline* with the call-stack of one process. The right side contains the *Function Summary* that provides a function profile and a *Communication Matrix* depicting a profile about the communication between the parallel processes. The trace is available for download at http://vampir.eu.

3.2.1 Scalable Trace Formats

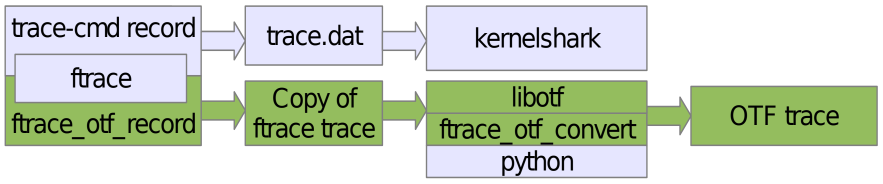
Scalable performance analysis tools commonly used by the HPC community make use of scalable formats such as OTF [17], OTF2 [9], CTF [4], and HPCTRACE [1]. The *Open Trace Format* (OTF) was designed for use with VampirTrace [24] and enables parallel reading and writing of trace files. The format is built around the concept of event streams, which can hold trace information of one or more parallel execution entities (processes, threads, GPU streams). Event properties, such as names of processes and functions as well as grouping information, can be defined locally for one stream or globally for all streams. This separation of different event streams as well as meta-data is important for efficiently reading and writing event traces in parallel, which has already been demonstrated on a massively parallel scale with more than 200,000 event streams [13]. The data itself is encoded in ASCII format and can be compressed transparently. The successor of this trace format is OTF2 [9]. It has a similar structure but allows for more efficient (binary) encoding and processing. OTF2 is part of the Score-P performance measurement environment [18].

We use Vampir for the visualization of the generated OTF files. Figure 1 shows the visualization of a trace of a typical MPI application recorded using VampirTrace. Vampir is designed to display the temporal relation between parallel processes as well as the behavior of individual processes, to present performance metrics, e.g., hardware counters, MPI communication and synchronization events. Additionally, Vampir derives profiling information from the trace, including a function summary, a communication matrix, and I/O statistics. Starting from an overall view on the trace data, Vampir enables the user to interactively browse through the trace data to find performance anomalies. By providing the capability of filtering the trace data, Vampir helps users to cope with potentially large amounts of trace data that has been recorded by the measurement infrastructure. Moreover, it provides a client-server based infrastructure using a parallel analysis server that can run on multiple nodes to interactively browse through the large amounts of trace data.

In general, the trace files can be written and read through an open source library to enable users to analyze the traces with custom tools. VampirTrace and OTF are



(a) Conversion of perf.data recordings



(b) Conversion of ftrace recordings

Figure 2: Toolchains for recording and converting performance data of Linux performance monitoring tools.

bundled with command-line tools for analyzing and processing OTF traces. Since the focus of these tools has been instrumentation based recording, there is no dedicated call-path sample record type in OTF or any of the other formats supported by Vampir so far. Therefore, the call-path sample information from perf.data is mapped to enter- and leave-function events typically obtained through instrumentation. Introducing support for sampled events into the full tool chain is currently work in progress.

3.2.2 Conversion of Trace Data

To convert the perf.data information into a scalable file format, we use the python interface provided by `perf script` and the python-bindings of OTF. Additionally, we patched perf script to pass dynamic symbol object information to the conversion script[3]. Based on the PID and CPU information within every sample, we are able to create two different traces: a task-centric and a CPU-centric trace. The conversion process depicted in Figure 2a is still sequential due to the limitations of the perf.data file format. For a CPU-centric view, this limitation could be overcome with multiple perf data files – one per CPU – which would be feasible with the existing tool infrastructure. However, task migration activities and their event presentation do pose a major challenge for a task-centric view since information on individual tasks would be scattered among multiple data files.

Note that perf.data information that is gathered in a task-specific context does not provide information about the CPU that issued a specific event. Thus, we can only create task-centric traces in this case. Information that is gathered in a CPU-specific context allows us to create both CPU-centric and task-centric traces.

Processing information provided by ftrace is straightforward as the exact enter and exit events are captured. Thus, we use the OTF python bindings to write events whenever a function is entered or exited. We concurrently generate two traces – a CPU-centric trace and a process-centric trace. If a function has been filtered out or the process has been unscheduled in between, enter events are written to match the current stack depth. One challenge for the trace generation is the timer resolution of ftrace events, which is currently in microseconds. This leads to a lower temporal accuracy within the traces as function call timer resolution is nanoseconds. The difference of these timer sources adds uncertainty. However, the order and context of the calls stay correct, thereby allowing enthusiasts to understand causal relations of function calls within the kernel. The full toolchain overview is depicted in Figure 2b.

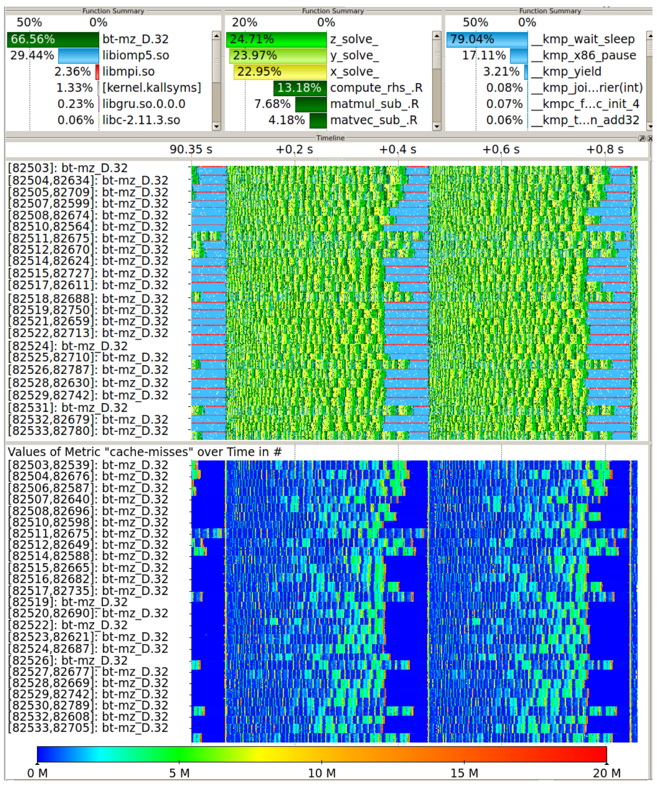
4 Examples and Results

4.1 Analyzing Parallel Scientific Applications

This example demonstrates the scalability of our approach. We use the perf-based tool infrastructure presented in Section 3 to analyze a hybrid parallel application. The target application is bt-mz of the NAS parallel benchmark suite [2] (Class D, 32 MPI processes with 8 OpenMP threads each).

We run and post-process this workload on a NUMA shared memory system with a total of 512 cores and

[3]See https://lkml.org/lkml/2014/2/18/57


Function Summary

66.56%	bt-mz_D.32
29.44%	libiomp5.so
2.36%	libmpi.so
1.33%	[kernel.kallsyms]
0.23%	libgru.so.0.0.0
0.06%	libc-2.11.3.so

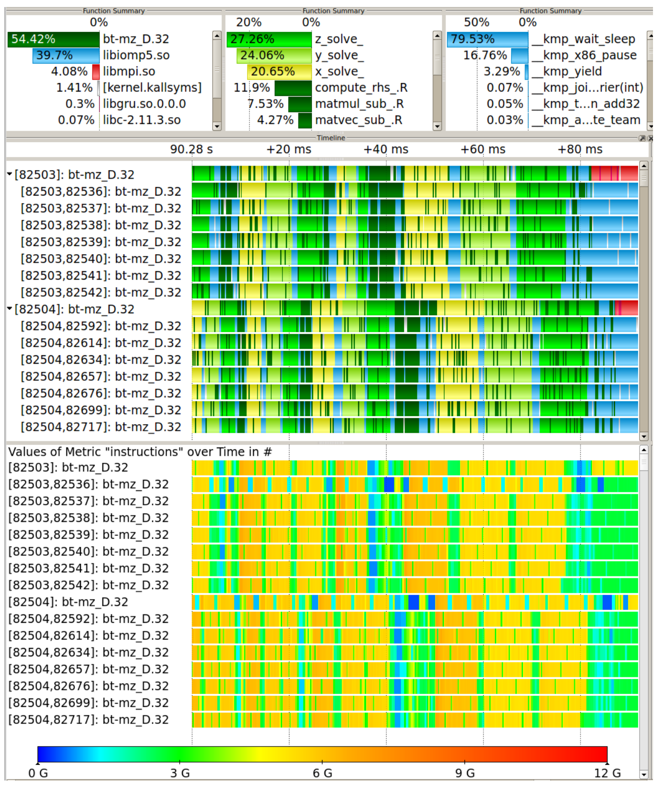
Function Summary

24.71%	z_solve_
23.97%	y_solve_
22.95%	x_solve_
13.18%	compute_rhs_.R
7.68%	matmul_sub_.R
4.18%	matvec_sub_.R

Function Summary

79.04%	__kmp_wait_sleep
17.11%	__kmp_x86_pause
3.21%	__kmp_yield
0.08%	__kmp_joi...rier(int)
0.07%	__kmpc_f...c_init_4
0.06%	__kmp_t...n_add32

(a) All processes, two iterations with cache-misses per second


Function Summary

54.42%	bt-mz_D.32
39.7%	libiomp5.so
4.08%	libmpi.so
1.41%	[kernel.kallsyms]
0.3%	libgru.so.0.0.0
0.07%	libc-2.11.3.so

Function Summary

27.26%	z_solve_
24.06%	y_solve_
20.65%	x_solve_
11.9%	compute_rhs_.R
7.53%	matmul_sub_.R
4.27%	matvec_sub_.R

Function Summary

79.53%	__kmp_wait_sleep
16.76%	__kmp_x86_pause
3.29%	__kmp_yield
0.07%	__kmp_joi...rier(int)
0.05%	__kmp_t...n_add32
0.03%	__kmp_a...te_team

(b) Two processes, close-up inside one iteration which instructions per second

Figure 3: Trace of a parallel application using OpenMP and MPI. Program execution is colored green/yellow. Thread synchronization via OpenMP is colored blue. Process synchronization via MPI is colored red.

8 TiB main memory[4]. To generate the trace, we only use features that are available for unprivileged users in standard Linux environments. We utilize `perf record` with default settings for the cycles, instructions, and cache-misses hardware events and enabled call-graph tracing.

Additional cores are reserved for perf to reduce the perturbation of the application due to the measurement. The recording operates at the limit of the system I/O capacity, so that a number of chunks are lost. According to internal measurements of the application, its execution time increases from 70.4 s to 95.8 s when comparing a regular and a measured execution. Considering the scale of the application and three hardware counters with a relatively high recording frequency, the overhead is acceptable. The resulting perf.data file contains 166 million events in 16 GiB. After the conversion process, the resulting compressed OTF trace has a size of 2.1 GiB.

[4]SGI UV2000 with 64 socket Intel Sandy Bridge E5-4650L @ 2.6 GHz

Figure 3a visualizes an excerpt of approx. 1 second of the application execution in Vampir. For a more concise visualization, we filter the shepherd threads of the OpenMP run-time library as well as the `mpirun` and helper processes. These tasks monitor the the OpenMP threads and the MPI environment for failures. They are recorded along the other tasks but do not show any regular activity during the execution. The figure contains three summaries of function activity: the fraction of time spent in each dso, the time share of functions in the binary, and the time share of functions in the OpenMP library. It also contains a timeline with the current function and a heat-map of cache-misses for all processes respectively. The visualization contains two iterations of the application execution. After each iteration, a global synchronization (red) between all MPI ranks is performed. The computation threads also synchronize (light blue) with their respective master threads. At the very beginning of each iteration, there is a short phase with a high cache miss rate after which the miss rate drops. Towards the end of each iteration, the cache miss rate also increases and so does the run-time of the repeated `x/y/z_solve` functions. A closer look inside an iteration is shown in Figure 3b, which is focused on two processes (16 compute threads total). Within each process, the `x/y/z_solve` and a few other functions are repeatedly executed with OpenMP synchronizations in between. Note that there is some sampling noise of other function calls within the `x/y/z_solve` that cannot be filtered due to imperfect call-path information. The performance radar shows that the functions `x/y/z_solve` have different typical instruction rates. Two threads (82536 and 82504) show regular drops in the instruction rate and similar drops in the cycles rate (not shown in the picture). This is likely due to them being preempted in favor of another task. As a consequence, the synchronization slows down the entire thread groups. Moreover, there is a regular diagonal pattern of short drops in the instruction rate. This is likely a result of OS-noise similar to the effects that we analyze in Section 4.4.

4.2 Analyzing the Behavior of a Web Server

In addition to analyzing one (parallel) application, perf can also be used for system analyses. To demonstrate these capabilities, we ran perf as a privileged user on a virtual machine running a private ownCloud[5] installation using the Apache2 webserver and a MySQL database. The virtual machine is hosted on a VMware installation and is provided with 2 cores and 4 GB of memory. The recording was done using the `-a` flag to enable system-wide recording in addition to call-graph sampling. The visualization of the resulting trace is shown in Figure 4. The recorded workload consisted of six WebDAV clients downloading 135 image files with a total size of 500 MB per client.

The parallel access of the clients is handled through the Apache2 `mpm_prefork` module, which maintains a pool of server processes and distributes requests to these workers. This is meant to ensure scalable request handling with a high level of separation between the workers and is recommended for PHP applications. The process pool can be configured with a minimum and maximum number of server processes based on the number of expected clients. However, the high load from the clients downloading files in parallel in conjunction with the small number of available cores leads to an overload that manifests itself through the parallel server processes spending much time in the `idle(R)` state in which processes are run-able and represented in the kernel's task queue but not actually running, e.g., they are not waiting for I/O operations to complete. These involuntary context switches are distinctive for overload situations and are also reflected by the high number of context switches, as can be seen in the display in the middle of the figure.

The MySQL database is involved in the processing as it stores information about the files and directories stored on the server. Every web-server instance queries the database multiple times for each client request. Since the run-times of the database threads between voluntary context switches (waiting for requests) are relatively short, the threads are not subject to involuntary switches.

In addition to the run-time behavior of the processes and their scheduling, we have also captured information about the network communication of the server. This is depicted in the lower three displays of Figure 4. To accomplish this, two additional events have been selected during the recording: `net:net_dev_xmit` reflecting the size of the socket buffers handed to the network device for transmission to clients and `net:netif_receive_skb` for received socket buffers. Note that this information does not necessarily reflect the exact data rate on the network but can provide a good estimate of the network load and how it can be attributed to different processes.

4.3 Analyzing Parallel Make Jobs

In addition to analyzing the performance of server workloads, `perf` can also be used to record the behavior of desktop machines. As an example, we use the compilation process of the perf project using the GCC 4.8.0 compiler. As in the previous example, `perf` has been run as a privileged user in order to capture scheduling and migration events in addition to the `cycles` and `page-faults` counter. Figure 5 shows the compilation process in four different configurations, from a serial build to a highly parallel build on a four core desktop machine (Intel Core i7-2620M). The serial compilation is depicted in Figure 5a and reveals that one compilation step requires significantly more time to finish than all other steps. Figure 5b depicts a parallel `make` to compensate for the wait time (and to better utilize the

[5] See http://owncloud.org/

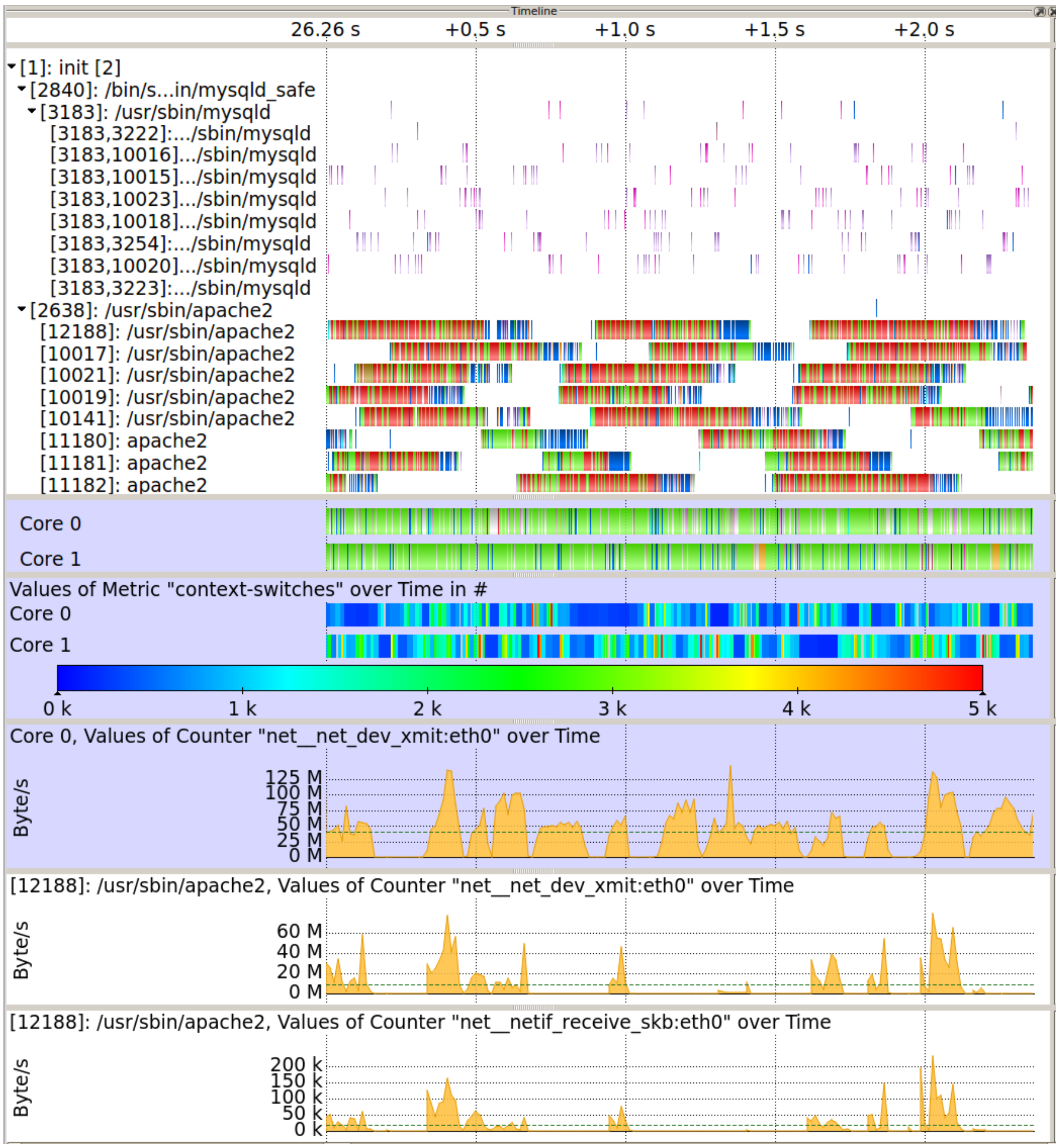
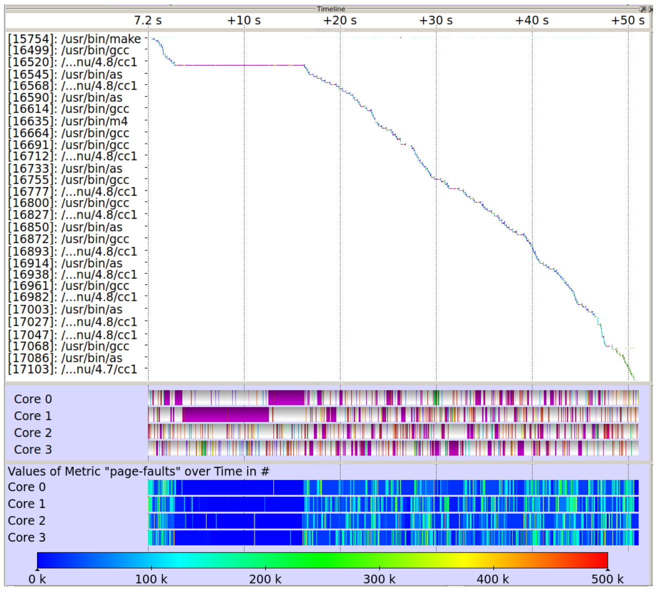
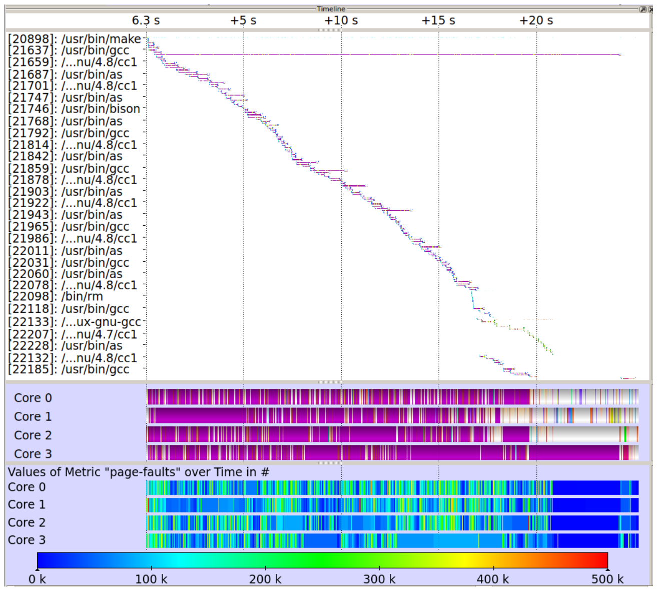

Timeline
26.26 s
+0.5 s
+1.0 s
+1.5 s
+2.0 s
[1]: init [2]
[2840]: /bin/s...in/mysqld_safe
[3183]: /usr/sbin/mysqld
[3183,3222]:.../sbin/mysqld
[3183,10016].../sbin/mysqld
[3183,10015].../sbin/mysqld
[3183,10023].../sbin/mysqld
[3183,10018].../sbin/mysqld
[3183,3254]:.../sbin/mysqld
[3183,10020].../sbin/mysqld
[3183,3223]:.../sbin/mysqld
[2638]: /usr/sbin/apache2
[12188]: /usr/sbin/apache2
[10017]: /usr/sbin/apache2
[10021]: /usr/sbin/apache2
[10019]: /usr/sbin/apache2
[10141]: /usr/sbin/apache2
[11180]: apache2
[11181]: apache2
[11182]: apache2
Core 0
Core 1
Values of Metric "context-switches" over Time in #
Core 0
Core 1
0 k
1 k
2 k
3 k
4 k
5 k
Core 0, Values of Counter "net__net_dev_xmit:eth0" over Time
Byte/s
125 M
100 M
75 M
50 M
25 M
0 M
[12188]: /usr/sbin/apache2, Values of Counter "net__net_dev_xmit:eth0" over Time
Byte/s
60 M
40 M
20 M
0 M
[12188]: /usr/sbin/apache2, Values of Counter "net__netif_receive_skb:eth0" over Time
Byte/s
200 k
150 k
100 k
50 k
0 k

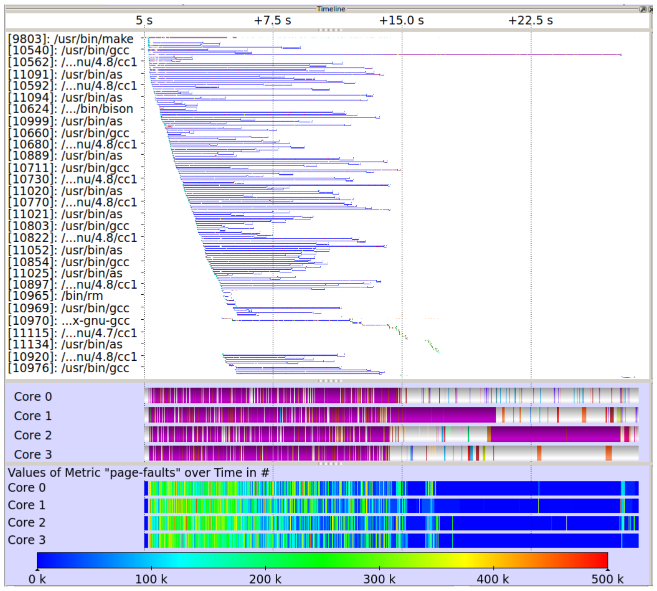
Figure 4: Vampir trace visualization of a system running an Apache2 web sever and a MySQL database. Some processes were filtered. The top display shows the thread-centric view followed by the CPU-centric view and the number of context switches per core. The lower part of the figure contains the average socket buffer size transmitted (`net_dev_xmit`) per time for core 0 and for one of the Apache2 processes as well as the average socket buffer size received per time by that process. The executed code parts are colored as follows: MySQL in purple, PHP5 in green, and `libc` in blue. For the cores, the function `native_safe_halt` is colored orange and is used on Core 1 when it is not needed toward the end. The `idle(R)` state is colored red.



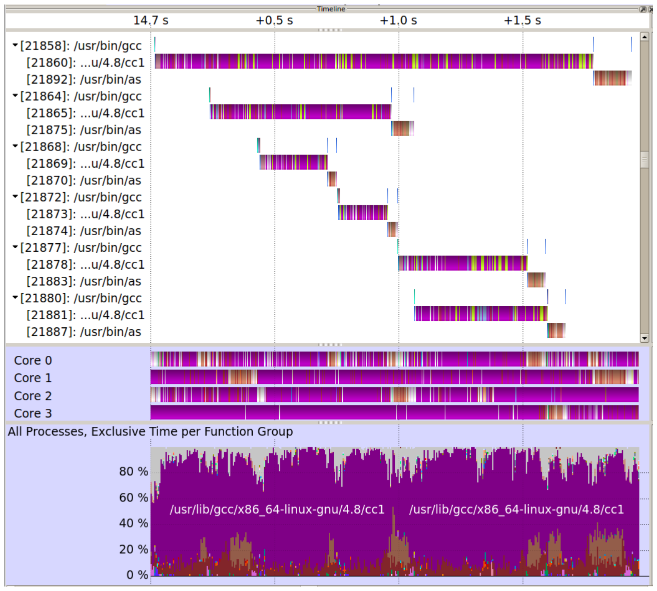
(a) serial `make`



(b) `make -j4`



(c) `make -j`



(d) `make -j4` (close-up)

Figure 5: System traces of a desktop machine compiling perf (as shipped with Linux 3.8.0) in different configurations: (a) serial make; (b) with four parallel make jobs (using `-j4`); (c) with unlimited number of make jobs (using `-j`); and (d) a close-up view of the trace shown in (b) showing six make jobs. Figures (a) – (c) each show both the process-centric view (top) and the CPU-centric view (bottom) of the function execution in addition to a display of the page faults that occurred during execution. Figure (d) also shows a summary of all executed processes during the depicted time-frame. The colors are as follows: `cc1` depicted in purple, `idle(R)` in blue, `as` in dark brown, `libc` in light brown, and kernel symbols in light gray. All figures only contain the compilation and linking steps, the preceding (sequential) configuration steps are left out intentionally.

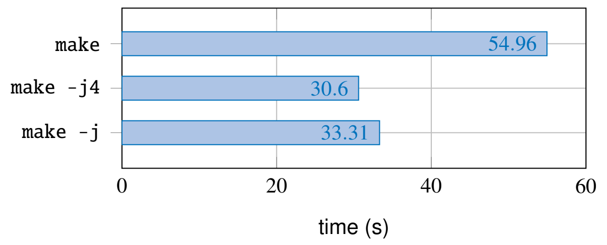


Figure 6: Time required for building the perf project using different configurations for parallel make (1, 4, unlimited).

available four CPU cores). It shows that the compilation proceeds even though the long running compilation step is not finished yet. Only at the very end, the linking step has to be deferred until all make jobs are finished. A subset of the parallel make steps is depicted in Figure 5d to visualize the process structure (`gcc` spawns the processes `cc` and `as` and waits for their execution to finish) and the actual parallel execution. The figure also shows the executed applications and library functions, e.g., `cc1`, `gcc`, `as`, and kernel symbols.

Another attempt to speed up the compilation (and to compensate for possible I/O idle times) is to spawn even more processes. This can be done using `make -j` without specifying the number of parallel jobs. In that case, `make` launches as many jobs as possible with respect to compilation dependencies. This can lead to heavy oversubscription even on multi-core systems, possibly causing a large number of context switches and other performance problems. The behavior of a highly parallel make is depicted in Figure 5c, which also shows an increased number of page faults as a consequence of the high number of context switches. Overall, compiling the perf project with `make -j4` is slightly faster (30.6 s) compared to using `make -j` (33.31 s), as shown in Figure 6.

4.4 Analyzing OS Behaviour with ftrace

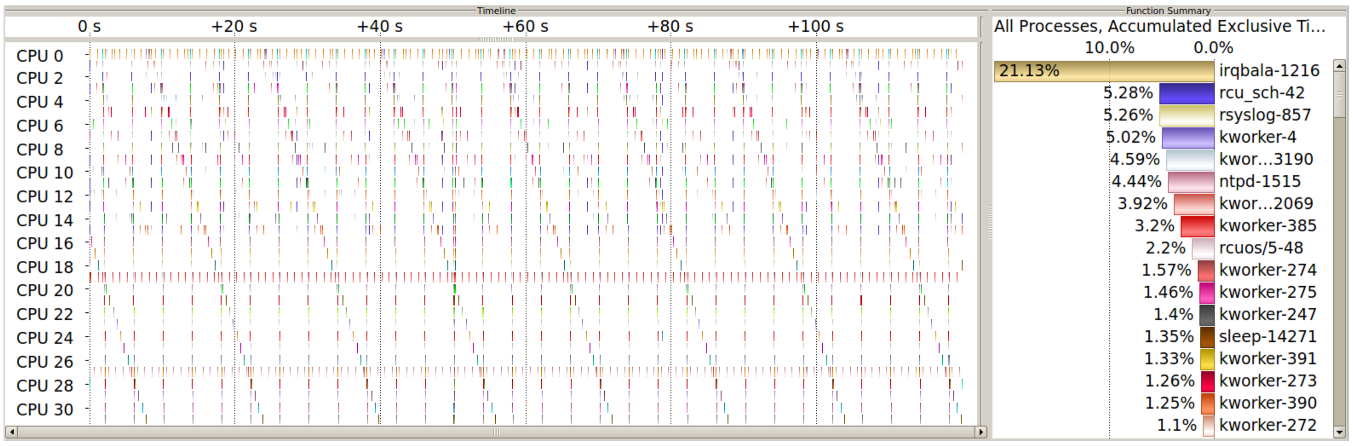
Figure 7a shows a trace of an idle dual socket system running Ubuntu Server 13.10. With eight cores per processor and HyperThreading active, 32 logical CPUs are available. We filtered out the idle functions that use up to 99.95 % of the total CPU time that is spent in the kernel. The largest remaining contributors are the irqbalancer, the RCU scheduler [21, 22], the rsyslog daemon, some kernel worker tasks, and the NTP daemon. We also see that there are two different kinds of per CPU threads that issue work periodically: watchdog threads and kernel worker threads that are used by the ondemand governor. Watchdog threads start their work every 4 s (displayed as vertical lines). The ondemand frequency governor is activated every 16 s on most CPUs (transversal lines). kworker-4 is the kernel worker thread of CPU 0. It uses significantly more time compared to other kernel workers since it is periodically activated by ksoftirq, which is running on CPU 0 and is handling IPMI messages at a regular interval of 1 s. CPU 22 also executes work every second triggered by the NTP daemon.

The RCU scheduler is mainly triggered by irqbalance and the rsyslog daemon. Zooming into the trace, we see that these tasks use the `__call_rcu` function. Shortly afterwards, the RCU scheduler starts and handles the grace periods of the RCU data. In this example, the RCU scheduler task runs on different processor cores but always on the same NUMA node as the process that issued RCU calls. Figure 7b depicts this behavior for the rsyslogd activity. After the RCU scheduler is migrated to another CPU, a kernel worker thread is scheduled. The kernel worker thread handles the ondemand frequency governor timer (od_dbs_timer, not depicted).

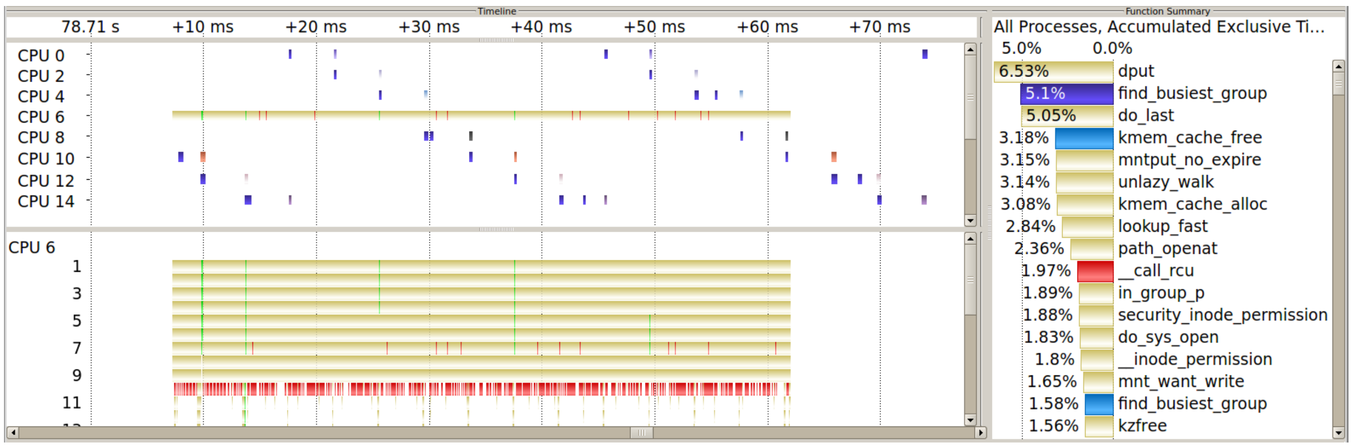
5 Conclusion and Future Work

This paper presents a new combined workflow for recording, managing, and visualizing performance data on Linux systems. We rely on established performance monitoring infrastructures and tools, making our approach applicable in a wide range of scenarios. Call-path sampling works on standard production systems and does not require root access or special permissions. Having additional permissions to record special tracepoint events can further increase the level of detail. By using already available Linux tools that require no recompilation or re-linking, the entry barrier for performance analysis has been lowered significantly. With sampling, the overhead can be controlled by selecting an appropriate event frequency. For a visual analysis, we leverage the Vampir visualization tool that originates from the HPC community. This enables a scalable and flexible visualization of trace data that contains information from a large number of processes, running over a long time, and including a high level of detail.

We have demonstrated the versatility of our approach with several use cases, including an analysis of scientific applications running on large production systems,



(a) Overview of kernel activity. Regular patterns: regular vertical lines every 4 seconds are watchdog threads; transversal lines every 16 seconds represent ondemand frequency governor activity.



(b) Zoomed into rsyslogd activity, which triggers RCU scheduler activity. rsyslogd calls to RCU objects are colored red. rsyslogd runs on CPU 6. The CPU location of the RCU scheduler changes over time across unoccupied cores of one NUMA package. The same NUMA package is used by rsyslogd and rcuos/6. The light blue activity (not depicted in timeline, but in function statistics) represents the rcuos/6 task that offloads RCU callbacks for CPU 6.

Figure 7: Kernel activity of an idle dual socket Intel Sandy Bridge node. Idle functions have been filtered out.

the activity on a highly utilized web and database server, as well as investigating operating system noise. Different performance aspects can be covered: Hardware performance counters, call-path samples, process and task management, library calls, and system calls provide a holistic view for post-mortem performance analysis, focusing either on the entire system or on a specific application. Given the pervasiveness of Linux, even more use cases are possible, for instance optimizing energy usage on mobile systems.

Our future work will focus on some remaining scalability issues. The recording process of perf record – where only one file is written – should be reconsidered as the number of CPUs will continue to increase. The conversion process to OTF should be re-implemented as it is currently single threaded. We provide kernel patches that add missing functionality to the existing tools rather than using the `perf_event_open` system call directly, as the latter would result in the re-implementation of several perf userspace tool features. Additionally, we submitted bug-fixes, one of which was accepted into the main-line kernel. Furthermore, we plan to integrate measurements on multiple nodes to generate a single sampling-based trace from a distributed application. This will allow us to study interactions between processes on different systems as in client-server scenarios and massively parallel applications. Moreover, we plan to switch to OTF2 as the successor of the OTF data format. OTF2 will include support for a sampling data type that will reduce the trace size and speed up the conversion process.

6 Acknowledgment

This work has been funded by the Bundesministerium für Bildung und Forschung via the research project CoolSilicon (BMBF 16N10186) and the Deutsche Forschungsgemeinschaft (DFG) via the Collaborative Research Center 912 "Highly Adaptive Energy-Efficient Computing" (HAEC, SFB 921/1 2011). The authors would like to thank Danny Rotscher, Thomas William, and all people who are involved in patching and developing perf and ftrace. You are great, keep up the good work!

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6), 2010.

[2] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks – summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[3] Jonathan Corbet. Fun with tracepoints. *LWN - Linux Weekly News - online*, August 2009.

[4] Mathieu Desnoyers. Common Trace Format (CTF) Specification (v1.8.2). *Common Trace Format GIT repository*, 2012.

[5] Advanced Micro Devices. *BIOS and Kernel Developer's Guide (BKDG) for AMD Family 15h Models 00h-0Fh Processors, Rev 3.08*, March 12, 2012.

[6] J. Dongarra, K. London, S. Moore, P. Mucci, and D. Terpstra. Using PAPI for Hardware Performance Monitoring on Linux Systems. In *Conference on Linux Clusters: The HPC Revolution*, Urbana, Illinois, June 2001.

[7] Jake Edge. A look at ftrace. *LWN - Linux Weekly News - online*, March 2009.

[8] Jake Edge. perfcounters being included into the mainline during the recently completed 2.6.31 merge window. *LWN - Linux Weekly News - online*, July 2009.

[9] Dominic Eschweiler, Michael Wagner, Markus Geimer, Andreas Knüpfer, Wolfgang E. Nagel, and Felix Wolf. Open Trace Format 2: The Next Generation of Scalable Trace Formats and Support Libraries. In Koen De Bosschere, Erik H. D'Hollander, Gerhard R. Joubert, David A. Padua, Frans J. Peters, and Mark Sawyer, editors, *PARCO*, volume 22 of *Advances in Parallel Computing*, pages 481–490. IOS Press, 2011.

[10] Pierre-Marc Fournier, Mathieu Desnoyers, and Michel R. Dagenais. Combined Tracing of the Kernel and Applications with LTTng. In *Proceedings of the 2009 Linux Symposium*, July 2009.

[11] Francis Giraldeau, Julien Desfossez, David Goulet, Mathieu Desnoyers, and Michel R. Dagenais. Recovering system metrics from kernel trace. In *Linux Symposium 2011*, June 2011.

[12] Sudhanshu Goswami. An introduction to KProbes. *LWN - Linux Weekly News - online*, April 2005.

[13] Thomas Ilsche, Joseph Schuchart, Jason Cope, Dries Kimpe, Terry Jones, Andreas Knüpfer, Kamil Iskra, Robert Ross, Wolfgang E Nagel, and Stephen Poole. Optimizing I/O forwarding techniques for extreme-scale event tracing. *Cluster Computing*, pages 1–18, 2013.

[14] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A, 3B, and 3C: System Programming Guide*, February 2014.

[15] James A. Keniston. Ptrace, Utrace, Uprobes: Lightweight, Dynamic Tracing of User Apps. In *Proceedings of the 2007 Linux Symposium*, June 2007.

[16] Namhyung Kim. perf tools: Introduce new 'ftrace' command, patch. *LWN - Linux Weekly News - online*, April 2013.

[17] Andreas Knüpfer, Ronny Brendel, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Introducing the Open Trace Format (OTF). In Vassil N. Alxandrov, Geert D. Albada, Peter M. A.

Sloot, and Jack J. Dongarra, editors, *6th International Conference on Computational Science (ICCS)*, volume 2, pages 526–533, Reading, UK, 2006. Springer.

[18] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen D. Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer S. Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, 2011, Dresden, Germany*. Springer, September 2012.

[19] John Levon. OProfile Internals. *OProfile online documentation*, 2003.

[20] John Levon. OProfile manual. *OProfile online documentation*, 2004.

[21] Paul E. McKenney. The new visibility of RCU processing. *LWN - Linux Weekly News - online*, October 2012.

[22] Paul E. McKenney and Jonathan Walpole. What is RCU, Fundamentally?. *LWN - Linux Weekly News - online*, December 2007.

[23] Philip J. Mucci, Shirley Browne, Christine Deane, and George Ho. PAPI: A Portable Interface to Hardware Performance Counters. In *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pages 7–10, 1999.

[24] Matthias S. Müller, Andreas Knüpfer, Matthias Jurenz, Matthias Lieber, Holger Brunst, Hartmut Mix, and Wolfgang E. Nagel. Developing Scalable Applications with Vampir, VampirServer and VampirTrace. In *Parallel Computing: Architectures, Algorithms and Applications*, volume 15. IOS Press, 2008.

[25] Steven Rostedt. trace-cmd: A front-end for Ftrace. *LWN - Linux Weekly News - online*, October 2010.

[26] Steven Rostedt. Using KernelShark to analyze the real-time scheduler. *LWN - Linux Weekly News - online*, February 2011.

[27] Sameer S. Shende and Allen D. Malony. The TAU Parallel Performance System. *Int. J. High Perform. Comput. Appl.*, 20(2), 2006.

[28] Mark Wielaard. A SystemTap update. *LWN - Linux Weekly News - online*, January 2009.

Veloces: An Efficient I/O Scheduler for Solid State Devices

Swapnil Pimpale
L3CUBE
pimpale.swapnil@gmail.com

Vishakha Damle
PICT
amogh.1337@gmail.com

Amogh Palnitkar
PICT
vishakha.22@gmail.com

Sarvesh Rangnekar
PICT
sarveshr7@gmail.com

Om Pawar
PICT
om.pawar1992@gmail.com

Nafisa Mandliwala
L3CUBE
nafisa.mandliwala@gmail.com

Abstract

Solid State Devices (SSD) use NAND-based Flash memory for storage of data. They have the potential to alleviate the ever-existing I/O bottleneck problem in data-intensive computing environments, due to their advantages over conventional Hard Disk Drives (HDD). SSDs differ from traditional mechanical HDDs in various respects. The SSDs have no moving parts and are thus free from rotational latency which dominates the disk access time of HDDs.

However, on the other hand, due to the long existence of HDDs as persistent storage devices, conventional I/O schedulers are largely designed for HDDs. They mitigate the high seek and rotational costs in mechanical disks through elevator-style I/O request ordering and anticipatory I/O. As a consequence, just by replacing conventional HDDs with SSDs in the storage systems without taking into consideration other properties like low latency, minimal access time and absence of rotary head, we may not be able to make the best use of SSDs.

We propose Veloces, an I/O scheduler which will leverage the inherent properties of SSDs. Since SSDs perform read I/O operations faster than write operations, Veloces is designed to provide preference to reads over writes. Secondly, Veloces implements optional front-merging of contiguous I/O requests. Lastly, writing in the same block of the SSD is faster than writing to different blocks. Therefore, Veloces bundles write requests belonging to the same block. Above implementation has shown to enhance the overall performance of SSDs for various workloads like File-server, Web-server and Mail-server.

1 Introduction

The SSDs are built upon semiconductors exclusively, and do not have moving heads and rotating platters like HDDs. Hence, they are completely free from the rotational latency which is responsible for the high disk access time of HDDs. This results in SSDs operational speed being one or two orders of magnitude faster than HDDs. However, on the other hand, due to the long existence of HDDs as persistent storage devices, existing I/O scheduling algorithms have been specifically designed or optimized based on characteristics of HDDs.

Current I/O schedulers in the Linux Kernel are designed to mitigate the high seek and rotational costs in mechanical disks. SSDs have many operational characteristics like low latency, minimal access time and absence of rotary head which need to be taken into account while designing I/O schedulers.

The rest of this paper is structured as follows. In Section 2, SSD characteristics are elaborated. In Section 3 the existing schedulers are studied and their characteristics and features are compared. In Section 4, we elaborate on the design details of the proposed scheduler, Veloces. Section 5 focuses on results and performance evaluation of our scheduler for different workloads. Finally, in Section 6 we conclude this paper. Section 7 covers the acknowledgments.

2 SSD Characteristics

2.1 Write Amplification

SSDs have been evolved from EEPROM (Electrically Erasable Programmable Read-Only Memory) which gives it distinctive properties. It consists of a number

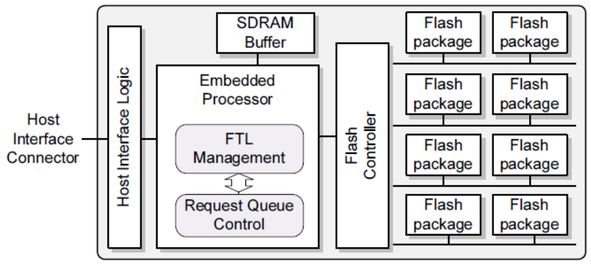


Figure 1: SSD Model

of blocks which can be erased independently. A block consists of pages. Read and write operations are performed at page level whereas erasing is done at block level. Overwriting is not allowed in SSDs, this makes the writes expensive. SSDs can only write to empty or erased pages. If it does not find any empty pages it finds an unused page and has to erase the entire block containing the page. Then it has to write the previous as well as the new page content on the block [10]. This makes SSDs slower over a period.

2.2 Garbage Collection

SSDs make use of Flash Translation Layer (FTL) to provide a mapping between the Logical Block Address (LBA) and the physical media [10]. FTL helps in improving the SSD performance by providing Wear Leveling and Garbage Collection. Wear Leveling helps in even distribution of data over the SSDs so that all the flash cells have same level of use. Garbage collection keeps track of unused or ‘stale’ pages and at an opportune moment erases the block containing good and stale pages, rewrites the good pages to another block so that the block is ready for further writes.

2.3 Faster Reads

Though SSDs provide a substantial improvement in I/O performance over the conventional HDDs, there is sufficient discrepancy between the read-write speeds. This is primarily due to the erase-before-write limitation. In Flash based devices it is necessary to erase a previously written location before overwriting to the said location. This problem is further aggravated by erase granularity which is much larger than the basic read/write granularity. As a result read operations in SSDs tend to be relatively faster than writes.

As mentioned earlier, SSDs do not possess rotary drive. Therefore, access times of I/O operations are relatively less affected by spatial locality of the request as compared to traditional HDDs. However it has been observed that the I/O requests in the same block tend to be slightly faster than I/O request in different block.

We have considered these features of SSDs while designing our scheduler.

3 Study of Existing Schedulers

In this section, we study the existing I/O schedulers and their drawbacks in case of SSD in the Linux Kernel 3.7.x and forward.

3.1 Noop Scheduler

The name Noop (No Operation) defines the working of this scheduler. It does not perform any kind of sorting or seek prevention, thus is the default scheduler for flash based devices where there is no rotary head. It performs minimum operations on the I/O requests before dispatching it to the underlying physical device [1].

The only chore that a NOOP Scheduler performs is merging, in which it coalesces the adjacent requests. Besides this it is truly a No Operation scheduler which merely maintains a request queue in FIFO order [7].

As a result, it is suitable for SSDs, which can be considered as random access devices. However this might not be true for all workloads.

3.2 Completely Fair Queuing Scheduler

CFQ scheduler attempts to provide fair allocation of the available disk I/O bandwidth for all the processes which requests an I/O operation.

It maintains per process queue for the processes which request I/O operation and then allocates time slices for each of the queues to access the disk [1]. The length of the time slice and the number of requests per queue depends on the I/O priority of the given process. The asynchronous requests are batched together in fewer queues and are served separately.

CFQ scheduler creates starvation of requests by assigning priorities. It has been primarily designed for HDD and does not consider the characteristics of SSDs while reordering the requests.

3.3 Deadline Scheduler

Deadline scheduler aims to guarantee a start service time for a request by imposing a deadline on all I/O operations.

It maintains two FIFO queues for read and write operations and a sorted Red Black Tree (RB Tree). The queues are checked first and the requests which have exceeded their deadlines are dispatched. If none of the requests have exceeded their deadline then sorted requests from RB Tree are dispatched.

The deadline scheduler provides an improved performance over Noop scheduler by attempting to minimize seek time and avoids starvation of the requests by imposing an expiry time for each request. It however performs reordering of requests according to their address which adds an extra overhead.

4 Proposed Scheduler - Veloces

In this section, we will discuss the implementation of our proposed scheduler - Veloces.

4.1 Read Preference

As mentioned earlier, Flash-based storage devices suffer from erase-before-write limitation. In order to overwrite to a previously known location, the said location must first be erased completely before writing new data. The erase granularity is much larger than the basic read granularity. This leads to a large read-write discrepancy [5][8]. Thus reads are considerably faster than writes.

For concurrent workloads with mixture of reads and writes the reads may be blocked by writes with substantial slowdown which leads to overall degradation in performance of the scheduler. Thus in order to address the problem of excessive reads blocked by writes, reads are given higher preference.

We have maintained two separate queues, a read queue and a write queue. The read requests are dispatched as and when they arrive. Each write request has an expiry time based on the incoming time of the request. The write requests are dispatched only when their deadline is reached or when there are no requests in the read queue.

4.2 Bundling of Write requests

In SSDs, it is observed that writes to the same logical block are faster than writes to different logical blocks. A penalty is incurred every time the block boundary is crossed [2][9]. Therefore, we have implemented bundling of write requests where write requests belonging to the same logical block are bundled together.

We have implemented this by introducing the buddy concept. A request X is said to be the buddy of request Y if the request Y is present in the same logical block as request X. For the current request, the write queue is searched for the buddy request. All such buddy requests are bundled together and dispatched.

Bundling count of these write requests can be adjusted according to the workloads to further optimize the performance.

4.3 Front Merging

Request A is said to be in front of request B when the starting sector number of request B is greater than the ending sector number of request A. Correspondingly, request A is said to be behind request B when the starting sector of request A is greater than the ending sector of request B.

The current I/O schedulers in the Linux Kernel facilitate merging of contiguous requests into a larger request before dispatch because serving a single large request is much more efficient than serving multiple small requests. However only back merging of I/O requests is performed in the Noop scheduler.

As SSDs do not possess rotational disks there is no distinction between backward and forward seeks. So, both front and back merging of the requests are employed by our scheduler.

5 Experimental Evaluation

5.1 Environment

We implemented our I/O Scheduler with parameters displayed in Table 1.

Type	Specifics
CPU/RAM	Intel Core 2 Duo 1.8GHz
SSD	Kingston 60GB
OS	Linux-Kernel 3.12.4 / Ext4 File System
Benchmark	Filebench Benchmark for Mail Server, Webserver and File Server workloads
Target	Our Scheduler and existing Linux I/O Schedulers

Table 1: System Specifications

5.2 Results

We used the FileBench benchmarking tool which generates workloads such as Mail Server, File Server and Webserver. The results of the benchmark are shown in Figure 2.

The graph shows Input/Output Operations performed per second (IOPS) by the four schedulers Noop, Deadline, CFQ and Veloces for the workloads Mail Server, File Server and Webserver. The Veloces scheduler performs better than the existing schedulers for all the tested workloads. It shows an improvement of up to 6% over the existing schedulers in terms of IOPS.

6 Conclusion

In conclusion, Flash-based storage devices are capable of alleviating I/O bottlenecks in data-intensive applications. However, the unique performance characteristics of Flash storage must be taken into account in order to fully exploit their superior I/O capabilities while offering fair access to applications.

Based on these motivations, we designed a new Flash I/O scheduler which contains three essential techniques to ensure fairness with high efficiency read preference, selective bundling of write requests and front merging of the requests.

We implemented the above design principles in our scheduler and tested it using FileBench as the benchmarking tool. The performance of our scheduler was consistent across various workloads like File Server, Web Server and Mail Server.

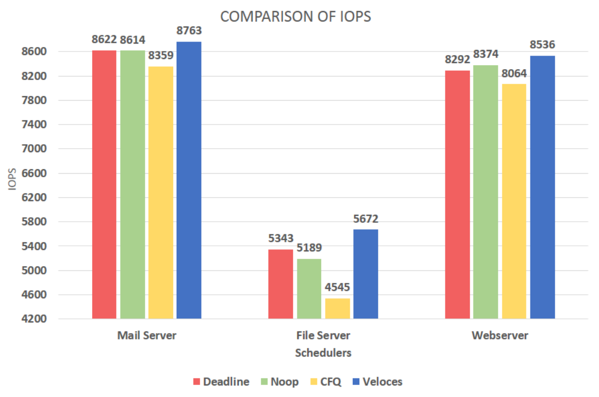


Figure 2: Comparison of IOPS

7 Acknowledgment

We would like to take this opportunity to thank our internal guide Prof. Girish Potdar, Pune Institute of Computer Technology for giving us all the help and guidance we needed. We are really grateful to him for his kind support throughout this analysis and design phase.

We are also grateful to other staff members of the Department for giving important suggestions.

Our external guides Mr Swapnil Pimpale and Ms Nafisa Mandliwala were always ready to extend their helping hand and share valuable technical knowledge about the subject with us.

We are thankful to PICT library and staff, for providing us excellent references, which helped us a lot to research and study topics related to the project.

References

[1] M.Dunn and A.L.N. Reddy, A new I/O scheduler for solid state devices. Tech. Rep. TAMU-ECE-2009-02, Department of Electrical and Computer Engineering, Texas A&M University, 2009.

[2] J. Kim, Y. Oh, E. Kim, J. Choi, D. Lee, and S.H. Noh, Disk schedulers for solid state drivers. In Proc. EMSOFT (2009), PP. 295-304.

[3] Wang H., Huang P., He S., Zhou K., Li C., and He X. A novel I/O scheduler for SSD with improved performance and lifetime. Mass Storage Systems (MSST), 2013 IEEE 29th Symposium, 1-5.

[4] S. Kang, H. Park, C. Yoo, Performance enhancement of I/O scheduler for solid state device. In 2011 IEEE International Conference on Consumer Electronics, 31-32.

[5] S. Park and K. Shen, *Fios: A fair, efficient flash i/o scheduler*, in FAST, 2012.

[6] Y. Hu, H. Jiang, L. Tian, H. Luo, and D. Feng, *Performance impact and interplay of ssd parallelism through advanced commands, allocation strategy and data granularity*, Proceedings of the 25th International Conference on Supercomputing (ICS'2011), 2011.

[7] J. Axboe. Linux block IO - present and future. In Ottawa Linux Symp., pages 51-61, Ottawa, Canada, July 2004.

[8] S. Park and K. Shen. A performance evaluation of scientific I/O workloads on flash-based SSDs. In IASDS'09:Workshop on Interfaces and Architectures for Scientific Data Storage, New Orleans, LA, Sept. 2009.

[9] J. Lee, S. Kim, H. Kwon, C. Hyun, S. Ahn, J. Choi, D. Lee, and S. H. Noh. Block recycling schemes and their cost-based optimization in NAND Flash memory based storage system. In EMSOFT'07: 7th ACM Conf. on Embedded Software, pages 174-182, Salzburg, Austria, Oct. 2007.

[10] S. Park , E. Seo , J. Shin , S. Maeng and J. Lee. Exploiting Internal Parallelism of Flash-based SSDs. In IEEE computer architecture letters, Vol. 9, No. 1, Jan–Jun 2010.

Dmdedup: Device Mapper Target for Data Deduplication

Vasily Tarasov
Stony Brook University & IBM Research
tarasov@vasily.name

Deepak Jain
Stony Brook University
dpakjain@gmail.com

Geoff Kuenning
Harvey Mudd College
geoff@cs.hmc.edu

Sonam Mandal
Stony Brook University
somandal@cs.stonybrook.edu

Karthikeyani Palanisami
Stony Brook University
karthikeyani.palanisami@gmail.com

Philip Shilane
EMC
philip.shilane@emc.com

Sagar Trehan
Stony Brook University
sagartrehan@gmail.com

Erez Zadok
Stony Brook University
ezk@fsl.cs.sunysb.edu

Abstract

We present *Dmdedup*, a versatile and practical primary-storage deduplication platform suitable for both regular users and researchers. Dmdedup operates at the block layer, so it is usable with existing file systems and applications. Since most deduplication research focuses on metadata management, we designed and implemented a flexible backend API that lets developers easily build and evaluate various metadata management policies. We implemented and evaluated three backends: an in-RAM table, an on-disk table, and an on-disk COW B-tree. We have evaluated Dmdedup under a variety of workloads and report the evaluation results here. Although it was initially designed for research flexibility, Dmdedup is fully functional and can be used in production. Under many real-world workloads, Dmdedup's throughput exceeds that of a raw block device by 1.5–6×.

1 Introduction

As storage demands continue to grow [2], even continuing price drops have not reduced total storage costs. Removing duplicate data (deduplication) helps this problem by decreasing the amount of physically stored information. Deduplication has often been applied to backup datasets because they contain many duplicates and represent the majority of enterprise data [18, 27]. In recent years, however, primary datasets have also expanded substantially [14], and researchers have begun to explore *primary* storage deduplication [13, 21].

Primary-storage deduplication poses several challenges compared to backup datasets: access locality is less pronounced; latency constraints are stricter; fewer duplicates are available (about 2× vs. 10× in backups); and the deduplication engine must compete with other processes for CPU and RAM. To facilitate research in primary-storage deduplication, we developed and here present a flexible and fully operational primary-storage deduplication system, *Dmdedup*, implemented in the Linux kernel. In addition to its appealing properties for regular users, it can serve as a basic platform both for experimenting with deduplication algorithms and for studying primary-storage datasets and workloads. In earlier studies, investigators had to implement their own deduplication engines from scratch or use a closed-source enterprise implementation [3, 21, 23]. Dmdedup is publicly available under the GPL and we submitted the code to the Linux community for initial review. Our final goal is the inclusion in the mainline distribution.

Deduplication can be implemented at the file system or block level, among others. Most previous primary-storage deduplication systems were implemented in the file system because file-system-specific knowledge was available. However, block-level deduplication does not require an elaborate file system overhaul, and allows any legacy file system (or database) to benefit from deduplication. Dmdedup is designed as a stackable Linux kernel block device that operates at the same layer as software RAID and the Logical Volume Manager (LVM). In this paper, we present Dmdedup's design, demonstrate its flexibility, and evaluate its performance, memory usage, and space savings under various workloads.

Most deduplication research focuses on metadata management. Dmdedup has a modular design that allows it to use different *metadata backends*—data structures for maintaining hash indexes, mappings, and reference counters. We designed a simple-to-use yet expressive API for Dmdedup to interact with the backends. We implemented three backends with different underlying data structures: an in-RAM hash table, an on-disk hash table, and a persistent Copy-on-Write B-tree. In this paper, we present our experiences and lessons learned while designing a variety of metadata backends, and include detailed experimental results. We believe that our results and open-source deduplication platform can significantly advance primary deduplication solutions.

2 Design

In this section, we classify Dmdedup's design, discuss the device-mapper framework, and finally present Dmdedup's architecture and its metadata backends.

2.1 Classification

Levels. Deduplication can be implemented at the *application*, *file system*, or *block* level. Applications can use specialized knowledge to optimize deduplication, but modifying every application is impractical.

Deduplication in the *file system* benefits many applications. There are three approaches: (1) modifying an existing file system such as Ext3 [15] or WAFL [21]; (2) creating a stackable deduplicating file system either in-kernel [26] or using FUSE [11,20]; or (3) implementing a new deduplicating file system from scratch, such as EMC Data Domain's file system [27]. Each approach has drawbacks. The necessary modifications to an existing file system are substantial and may harm stability and reliability. Developing in-kernel stackable file systems is difficult, and FUSE-based systems perform poorly [19]. A brand-new file system is attractive but typically requires massive effort and takes time to reach the stability that most applications need. Currently, this niche is primarily filled by proprietary products.

Implementing deduplication at the *block* level is easier because the block interface is simple. Unlike many file-system-specific solutions, block-level deduplication can be used beneath any block-based file system such as Ext4, GPFS, BTRFS, GlusterFS, etc., allowing researchers to bypass a file system's limitations and design their own block-allocation policies. For that reason, we chose to implement Dmdedup at the block level. Our design means that Dmdedup can also be used with databases that require direct block-device access.

The drawbacks of block-level deduplication are threefold: (1) it must maintain an extra mapping (beyond the file system's map) between logical and physical blocks; (2) useful file-system and application context is lost; and (3) variable-length chunking is more difficult at the block layer. Dmdedup provides several options for maintaining logical-to-physical mappings. In the future, we plan to recover some of the lost context using file system and application hints.

Timeliness. Deduplication can be performed *in-line* with incoming requests or *off-line* via background scans. In-line deduplication saves bandwidth by avoiding repetitive reads and writes on the storage device and permits deduplication on a busy system that lacks idle periods. But it risks negatively impacting the performance of primary workloads. Only a few studies have addressed this issue [21,24]. Dmdedup performs inline deduplication; we discuss its performance in Section 4.

2.2 Device Mapper

The Linux Device Mapper (*DM*) framework, which has been part of mainline Linux since 2005, supports stackable block devices. To create a new device type, one builds a *DM target* and registers it with the OS. An administrator can then create corresponding *target instances*, which appear as regular block devices to the upper layers (file systems and applications). Targets rest above one or more physical devices (including RAM) or lower targets. Typical examples include software RAID, the Logical Volume Manager (LVM), and encrypting disks. We chose the DM framework for its performance and versatility: standard, familiar tools manage DM targets. Unlike user-space deduplication solutions [11, 13, 20] DM operates in the kernel, which improves performance yet does not prohibit communication with user-space daemons when appropriate [19].

2.3 Dmdedup Components

Figure 1 depicts Dmdedup's main components and a typical setup. Dmdedup is a stackable block device that rests on top of physical devices (e.g., disk drives, RAIDs, SSDs), or stackable ones (e.g., encryption DM

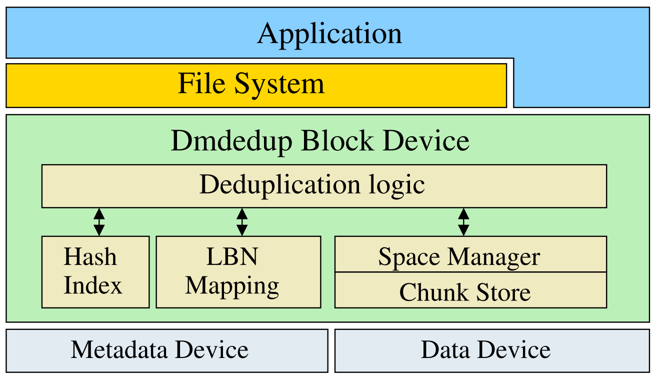


Figure 1: Dmdedup high-level design.

target). This approach provides high configurability, which is useful in both research and production settings.

Dmdedup typically requires two block devices to operate: one each for *data* and *metadata*. Data devices store actual user information; metadata devices track the deduplication metadata (e.g., a hash index). Dmdedup can be deployed even if a system has only one storage device, simply by creating two partitions. Although any combination of data and metadata devices can be used, we believe that using an HDD for data and an SSD for metadata is practical in a modern system. Deduplication metadata sizes are much smaller than the data size—often less than 1% of the data—but metadata is critical enough to require low-latency access. This combination matches well with today's SSD size and performance characteristics, and ties into the growing trend of combining disk drives with a small amount of flash.

To upper layers, Dmdedup provides a conventional block interface: reads and writes with specific sizes and offsets. Every write to a Dmdedup instance is checked against all previous data. If a duplicate is detected, the corresponding metadata is updated and no data is written. Conversely, a write of new content is passed to the data device and tracked in the metadata.

Dmdedup main components are (Figure 1): (1) *deduplication logic* that chunks data, computes hashes, and coordinates other components; (2) a *hash index* that tracks the hashes and locations of the chunks; (3) a *mapping* between Logical Block Numbers (LBNs) visible to upper layers and the Physical Block Numbers (PBNs) where the data is stored; (4) a *space manager* that tracks space on the data device, maintains reference counts, allocates new blocks, and reclaims unreferenced data; and (5) a *chunk store* that saves user data to the data device.

2.4 Write Request Handling

Figure 2 shows how Dmdedup processes write requests.

Chunking. The deduplication logic first splits all incoming requests into aligned, *subrequests* or chunks with a configurable power-of-two size. Smaller chunks allow Dmdedup to detect more duplicates but increase the amount of metadata [14], which can harm performance because of the higher metadata cache-miss rate. However, larger chunks can also hurt performance because they can require *read-modify-write* operations. To achieve optimal performance, we recommend that Dmdedup's chunk size should match the block size of the file system above. In our evaluation we used 4KB chunking, which is common in many modern file systems.

Dmdedup does not currently support Content-Defined Chunking (CDC) [16] although the feature could be added in the future. We believe that CDC is less viable for inline primary-storage block-level deduplication because it produces a mismatch between request sizes and the underlying block device, forcing a read-modify-write operation for most write requests.

After chunking, Dmdedup passes subrequests to a pool of working threads. When all subrequests originating from an initial request have been processed, Dmdedup notifies the upper layer of I/O completion. Using several threads leverages multiple CPUs and allows I/O wait times to overlap with CPU processing (e.g., during some hash lookups). In addition, maximal SSD and HDD throughput can only be achieved with multiple requests in the hardware queue.

Hashing. For each subrequest, a worker thread first computes the hash. Dmdedup supports over 30 hash functions from the kernel's crypto library. Some are implemented using special hardware instructions (e.g., *SPARC64 crypt* and *Intel SSE3* extensions). As a rule, deduplication hash functions should be collision-resistant and cryptographically strong, to avoid inadvertent or malicious data overwrites. Hash sizes must be chosen carefully: a larger size improves collision resistance but increases metadata size. It is also important that the chance of a collision is significantly smaller than the probability of a disk error, which has been empirically shown to be in the range 10^{-18}–10^{-15} [9]. For

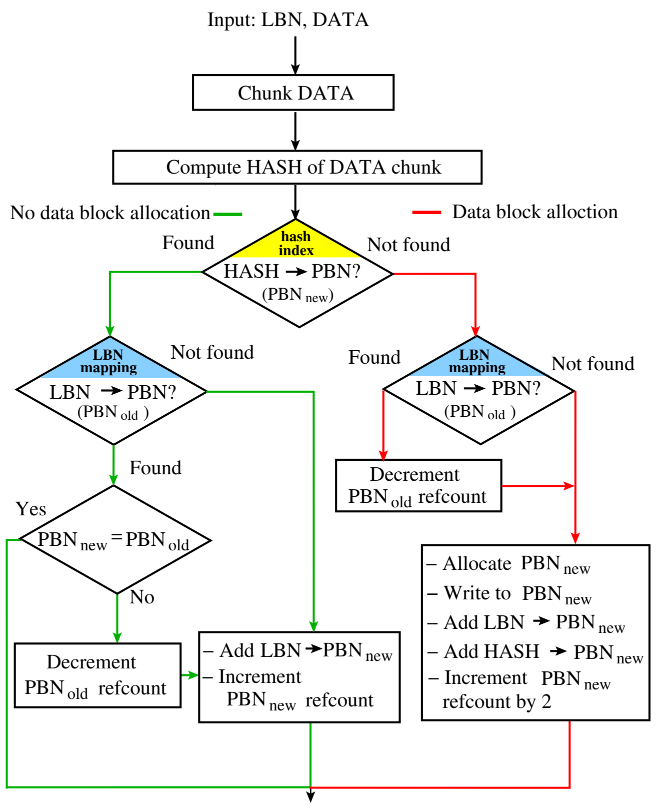


Figure 2: Dmdedup write path. PBN_{new} is the PBN found in the hash index: a new physical location for incoming data. PBN_{old} is the PBN found in the LBN mapping: the old location of data, before the write ends.

128-bit hashes, the probability of collision is less than 10^{-18} as long as the number of unique chunks is less than 2.6×10^{10}. For 4KB chunking, this corresponds to almost 100TB of unique data. Assuming a primary-storage deduplication ratio of 2× [12], Dmdedup can support up to 200TB of logical space in such configuration. In our experiments we used 128-bit MD5 hashes.

Hash index and LBN mapping lookups. The main deduplication logic views both the hash index and the LBN mapping as abstract key-value stores. The hash index maps hashes to 64-bit PBNs; the LBN map uses the LBN as a key to look up a 64-bit PBN. We use PBN_{new} to denote the value found in the hash index and PBN_{old} for the value found in the LBN mapping.

Metadata updates. Several cases must be handled; these are represented by branches in Figure 2. First, the hash might be found in the index (left branch), implying that the data already exists on disk. There are two sub-cases, depending on whether the target LBN exists in the LBN→PBN mapping. If so, and if the corresponding PBNs are equal, the upper layer overwrote a location (the LBN) with data that was already there; this is surprisingly common in certain workloads [13]. If the LBN is known but mapped to a different PBN, then the data on the LBN must have changed; this is detected because the hash-to-PBN mapping is one-to-one, so PBN_{old} serves as a proxy for a different hash. Dmdedup decrements PBN_{old}'s reference count, adds the LBN→PBN_{new} mapping, and increments PBN_{new}'s reference count. On the other hand, if the hash→PBN mapping is found but the LBN→PBN one is not (still on the left side of the flowchart), we have a chunk of data that has been seen before (i.e., a duplicate) being written to a previously unknown LBN. In this case we add a LBN→PBN_{new} mapping and increment PBN_{new}'s reference count.

The flowchart's right side deals with the case where the hash is not found: a data chunk hasn't been seen before. If the LBN is also new (right branch of the right side), we proceed directly to allocation. If it is not new, we are overwriting an existing block with new data, so we must first dereference the data that used to exist on that LBN (PBN_{old}). In both cases, we now allocate and write a PBN to hold the new data, add hash→PBN and LBN→PBN mappings, and update the reference counts. We increment the counts by two in these cases because PBNs are referenced from both hash index *and* LBN mapping. For PBNs that are referenced only from the hash index (e.g., due to LBN overwrite) reference counts are equal to one. Dmdedup decrements reference counts to zero during garbage collection.

Garbage collection. During overwrites, Dmdedup does not reclaim blocks immediately, nor does it remove the corresponding hashes from the index. This approach decreases the latency of the critical write path. Also, if the same data is rewritten after a period of non-existence (i.e., it is not reachable through the LBN mapping), it can still be deduplicated; this is common in certain workloads [17]. However, data-device space must eventually be reclaimed. We implemented an offline garbage collector that periodically iterates over all data blocks and recycles those that are not referenced.

If a file is removed by an upper-layer file system, the corresponding blocks are no longer useful. However, Dmdedup operates at the block layer and thus is unaware of these inaccessible blocks. Some modern file systems (e.g., Ext4 and NTFS) use the SATA TRIM command to inform SSDs that specific LBNs are not referenced anymore. Dmdedup takes advantage of these TRIMs to reclaim unused file system blocks.

2.5 Read Request Handling

In Dmdedup, reads are much simpler to service than writes. Every incoming read is split into chunks and queued for processing by worker threads. The LBN→PBN map gives the chunk's physical location on the data device. The chunks for the LBNs that are not in the LBN mapping are filled with zeroes; these correspond to reads from an offset that was never written. When all of a request's chunks have been processed by the workers, Dmdedup reports I/O completion.

2.6 Metadata Backends

We designed a flexible API that abstracts metadata management away from the main deduplication logic. Having pluggable metadata backends facilitates easier exploration and comparison of different metadata management policies. When constructing a Dmdedup target instance, the user specifies which backend should be used for this specific instance and passes the appropriate configuration parameters. Our API includes ten mandatory and two optional methods—including basic functions for initialization and destruction, block allocation, lookup, insert, delete, and reference-count manipulation. The optional methods support garbage collection and synchronous flushing of the metadata.

An unusual aspect of our API is its two types of key-value stores: *linear* and *sparse*. Dmdedup uses a linear store (from zero to the size of the Dmdedup device) for LBN mapping and a sparse one for the hash index. Backend developers should follow the same pattern, using the sparse store for key spaces where the keys are uniformly distributed. In both cases the interface presented to the upper layer after the store has been created is uniform: **kvs_insert**, **kvs_lookup**, and **kvs_delete**.

When designing the metadata backend API, we tried to balance flexibility with simplicity. Having more functions would burden the backend developers, while fewer functions would assume too much about metadata management and limit Dmdedup's flexibility. In our experience, the API we designed strikes the right balance between complexity and flexibility. During the course of the project, several junior programmers were asked to develop experimental backends for Dmdedup; anecdotally, they were able to accomplish their task in a short time and without changing the API.

We consolidated key-value stores, reference counting, and block allocation facilities within a single metadata backend object because they often need to be managed together and are difficult to decouple. In particular, when metadata is flushed, all of the metadata (reference counters, space maps, key-value stores) needs to be written to the disk at once. For backends that support transactions this means that proper ordering and atomicity of all metadata writes are required.

Dmdedup performs 2–8 metadata operations for each write. But depending on the metadata backend and the workload properties, every metadata operation can generate zero to several I/O requests to the metadata device.

Using the above API, we designed and implemented three backends: INRAM (in RAM only), DTB (disk table), and CBT (copy-on-write B-tree). These backends have significantly different designs and features; we detail each backend below.

2.6.1 INRAM Backend

INRAM is the simplest backend we implemented. It stores all deduplication metadata in RAM and consequently does not perform any metadata I/O. All *data*, however, is still stored on the data device as soon as the user's request arrives (assuming it is not a duplicate). INRAM metadata can be written persistently to a user-specified file at any time (e.g., before shutting the machine down) and then restored later. This facilitates experiments that should start with a pre-defined metadata state (e.g., for evaluating the impact of LBN space fragmentation). The INRAM backend allows us to determine the baseline of maximum deduplication performance on a system with a given amount of CPU power. It can also be used to quickly identify a workload's deduplication ratio and other characteristics. With the advent of DRAM backed by capacitors or batteries, this backend can become a viable option for production.

INRAM uses a statically allocated hash table for the sparse key-value store, and an array for the linear store. The linear mapping array size is based on the Dmdedup target instance size. The hash table for the sparse store is allocated (and slightly over-provisioned) based on the size of the data device, which dictates the maximum possible number of unique blocks. We resolve collisions with linear probing; according to standard analysis the default over-provisioning ratio of 10% lets Dmdedup complete a successful search in an average of six probes when the data device is full.

We use an integer array to maintain reference counters and allocate new blocks sequentially using this array.

2.6.2 DTB Backend

The disk table backend (DTB) uses INRAM-like data structures but keeps them on persistent storage. If no buffering is used for the metadata device, then every lookup, insert, and delete operation causes one extra I/O, which significantly harms deduplication performance. Instead, we use Linux's *dm-bufio* subsystem, which buffers both reads and writes in 4KB units and has a user-configurable cache size. By default, dm-bufio flushes all dirty buffers when more than 75% of the buffers are dirty. If there is no more space in the cache for new requests, the oldest blocks are evicted one by one. Dm-bufio also normally runs a background thread that evicts all buffers older than 60 seconds. We disabled this thread for deduplication workloads because we prefer to keep hashes and LBN mapping entries in the cache as long as there is space. The dm-bufio code is simple (1,100 LOC) and can be easily modified to experiment with other caching policies and parameters.

The downside of DTB is that it does not scale with the size of deduplication metadata. Even when only a few hashes are in the index, the entire on-disk table is accessed uniformly during hash lookup and insertion. As a result, hash index blocks cannot be buffered efficiently even for small datasets.

2.6.3 CBT Backend

Unlike the INRAM and DTB backends, CBT provides true transactionality. It uses Linux's on-disk Copy-On-Write (COW) B-tree implementation [6, 22] to organize

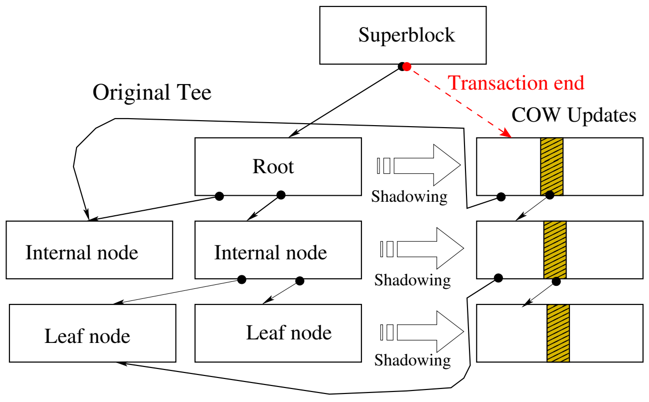


Figure 3: Copy-on-Write (COW) B-trees.

its key-value stores (see Figure 3). All keys and values are stored in a B^+-tree (i.e., values are located only in leaves). When a new key is added, the corresponding leaf must be updated. However, the COW B-tree does not do so in-place; instead, it *shadows* the block to a different location and applies the changes there. Next, the internal nodes of the B-tree are updated to point to the new leaf, again using shadowing. This procedure continues up to the root, which is referenced from a pre-defined location on the metadata device—the *Dmdedup superblock*. Multiple changes are applied to the B-tree in COW fashion but the superblock is not updated until Dmdedup explicitly ends the transaction. At that point, the superblock is atomically updated to reference the new root. As a result, if a system fails in the middle of a transaction, the user sees old data but not a corrupted device state. The CBT backend also allocates data blocks so that data overwrites do not occur within the transaction; this ensures both data and metadata consistency.

Every key lookup, insertion, or deletion requires $\log_b N$ I/Os to the metadata device (where b is the branching factor and N is the number of keys). The base of the logarithm is large because many key-pointer entries fit in a single 4KB non-leaf node (approximately 126 for the hash index and 252 for the LBN mapping). To improve CBT's performance we use dm-bufio to buffer I/Os. When a transaction ends, dm-bufio's cache is flushed. Users can control the length of a transaction in terms of the number of writes.

Because CBT needs to maintain intermediate B-tree nodes, its metadata is larger than DTB's. Moreover, the COW update method causes two copies of the updated blocks to reside in the cache simultaneously. Thus, for a given cache size, CBT usually performs more I/O than

DTB. But CBT scales well with the amount of metadata. For example, when only a few hashes are in the index, they all reside in one block and can be easily cached.

Statistics. Dmdedup collects statistics on the number of reads and writes, unique and duplicated writes, overwrites, storage and I/O deduplication ratios, and hash index and LBN mapping sizes, among others. These statistics were indispensable in analyzing our own experiments (Section 4).

Device size. Most file systems are unable to dynamically grow or shrink in response to changing device size. Thus, users must currently specify the Dmdedup device's size at construction time. However, the device's logical size depends on the actual deduplication ratio, which changes dynamically. Some studies offer techniques to predict dataset deduplication ratios [25], and we provide a set of tools to compute deduplication ratios in an existing dataset. If the deduplication ratio ever falls below expectations and the free data space nears to zero, Dmdedup warns the user via the OS's console or system log. The user can then remove unnecessary files from the device and force an immediate garbage collection, or add more data devices to the pool.

3 Implementation

The latest Dmdedup code has been tested against Linux 3.14 but we performed experimental evaluation on version 3.10.9. We kept the code base small to facilitate acceptance to the mainline and to allow users to investigate new ideas easily. Dmdedup's core has only 1,400 lines of code; the INRAM, DTB, and CBT backends have 500, 1,400, and 600 LOC, respectively. Over the course of two years, fifteen developers of varying skill levels have worked on the project. Dmdedup is open-source and was submitted for initial review to `dm-devel@` mailing list. The code is also available at `git://git.fsl.cs.sunysb.edu/linux-dmdedup.git`.

When constructing a Dmdedup instance, the user specifies the data and metadata devices, metadata backend type, cache size, hashing algorithm, etc. In the future, we plan to select or calculate reasonable defaults for most of these parameters. Dmdedup exports deduplication statistics and other information via the device mapper's STATUS `ioctl`, and includes a tool to display these statistics in a human-readable format.

4 Evaluation

In this section, we first evaluate Dmdedup's performance and overheads using different backends and under a variety of workloads. Then we compare Dmdedup's performance to Lessfs [11]—an alternative, popular deduplication system.

4.1 Experimental Setup

In our experiments we used three identical Dell PowerEdge R710 machines, each equipped with an Intel Xeon E5540 2.4GHz 4-core CPU and 24GB of RAM. Using lmbench we verified that the performance of all machines was within 4% of each other. We used an Intel X25-M 160GB SSD as the metadata device and a Seagate Savvio 15K.2 146GB disk drive for the data. Although the SSD's size is 160GB, in all our experiments we used 1.5GB or less for metadata. Both drives were connected to their hosts using Dell's PERC 6/i Integrated controller. On all machines, we used CentOS Linux 6.4 x86_64, upgraded to Linux 3.10.9 kernel. Unless otherwise noted, every experiment lasted from 10 minutes (all-duplicates data write) to 9 hours (Mail trace replay). We ran all experiments at least three times. Using Student's t distribution, we computed 95% confidence intervals and report them on all bar graphs; all half-widths were less than 5% of the mean.

We evaluated four setups: the raw device, and Dmdedup with three backends—INRAM, disk table (DTB), and COW B-Tree (CBT). In all cases Dmdedup's logical size was set to the size of the data device, 146GB, which allowed us to conduct experiments with unique data without running out of physical space. 146GB corresponds to 1330MB of metadata: 880MB of hash$\rightarrow$PBN entries (24B each), 300MB of LBN$\rightarrow$PBN entries (8B each), and 150MB of reference counters (4B each). We used six different metadata cache sizes: 4MB (0.3% of all metadata), 330MB (25%), 660MB (50%), 990MB (75%), 1330MB (100%), and 1780MB (135%). A 4MB cache corresponds to a case when no significant RAM is available for deduplication metadata; the cache acts as a small write buffer to avoid doing I/O with every metadata update. As described in Section 2.6.2, by default Dmdedup flushes dirty blocks after their number exceeds 75% of the total cache size; we did not change this parameter in our experiments. When the cache size is set to 1780MB (135% of all metadata) the 75% threshold equals to 1335MB, which is greater than the total size of all metadata (1330MB). Thus, even if all metadata is dirty, the 75% threshold cannot be reached and flushing never happens.

4.2 Dmdedup

To users, Dmdedup appears as a regular block device. Using its basic performance characteristics—sequential and random read/write behavior—one can estimate the performance of a complete system built using Dmdedup. Thus, we first evaluated Dmdedup's behavior with micro-workloads. To evaluate its performance in real deployments, we then replayed three production traces.

4.2.1 Micro-workloads

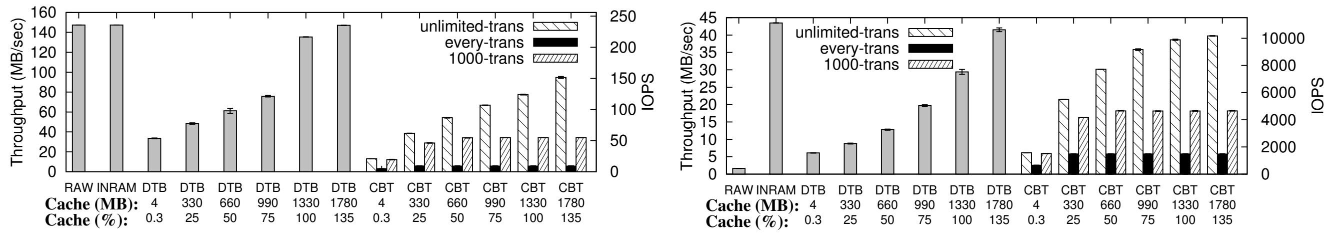
Unlike traditional (non-deduplicating) storage, a deduplication system's performance is sensitive to data content. For deduplication systems, a key content characteristic is its deduplication ratio, defined as the number of logical blocks divided by the number of blocks physically stored by the system [7]. Figures 4, 5, and 6 depict write throughput for our *Unique*, *All-duplicates*, and *Linux-kernels* datasets, respectively. Each dataset represents a different point along the content-redundancy continuum. *Unique* contains random data obtained from Linux's */dev/urandom* device. *All-duplicates* consists of a random 4KB block repeated for 146GB. Finally, *Linux-kernels* contains the sources of 40 Linux kernels (more details on the format follow).

We experimented with two types of micro-workloads: large sequential writes (subfigures (a) in Figures 4–6) and small random writes (subfigures (b) in Figures 4–6). I/O sizes were 640KB and 4KB for sequential and random writes, respectively. We chose these combinations of I/O sizes and access patterns because real applications tend either to use a large I/O size for sequential writes, or to write small objects randomly (e.g., databases) [5]. 4KB is the minimal block size for modern file systems. Dell's PERC 6/i controller does not support I/O sizes larger than 320KB, so large requests are split by the driver; the 640KB size lets us account for the impact of splitting. We started all experiments with an empty Dmdedup device and ran them until the device became full (for the Unique and All-kernels datasets) or until the dataset was exhausted (for Linux-kernels).

Unique (Figure 4). Unique data produces the lowest deduplication ratio (1.0, excluding metadata space). In this case, the system performs at its worst because the deduplication steps need to be executed as usual (e.g., index insert), yet all data is still written to the data device. For the sequential workload (Figure 4(a)), the INRAM backend performs as well as the raw device—147MB/sec, matching the disk's specification. This demonstrates that the CPU and RAM in our system are fast enough to do deduplication without any visible performance impact. However, it is important to note that CPU utilization was as high as 65% in this experiment.

For the DTB backend, the 4MB cache produces 34MB/sec throughput—a 75% decrease compared to the raw device. Metadata updates are clearly a bottleneck here. Larger caches improve DTB's performance roughly linearly up to 147MB/sec. Interestingly, the difference between DTB-75% and DTB-100% is significantly more than between other sizes because Dmdedup with 100% cache does not need to perform any metadata reads (though metadata writes are still required). With a 135% cache size, even metadata writes are avoided, and hence DTB achieves INRAM's performance.

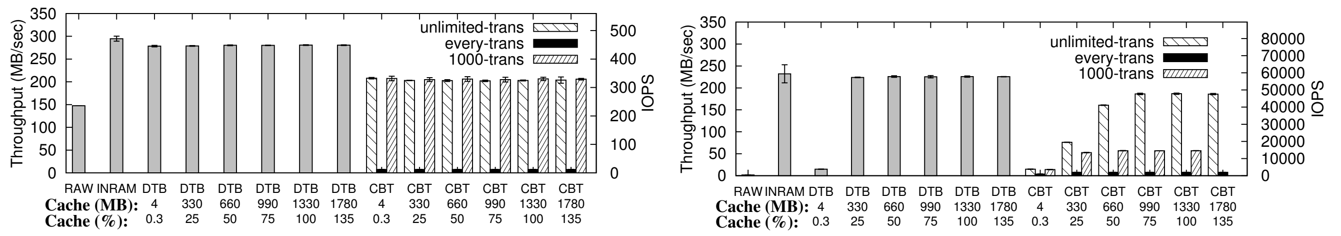
The CBT backend behaves similarly to DTB but its performance is always lower—between 3–95MB/sec depending on the cache and transaction size. The reduced performance is caused by an increased number of I/O operations, 40% more than DTB. The transaction size significantly impacts CBT's performance; an unlimited transaction size performs best because metadata flushes occur only when 75% of the cache is dirty. CBT with a transaction flush after every write has the worst performance (3–6MB/sec) because every write to Dmdedup causes an average of 14 writes to the metadata device: 4 to update the hash index, 3 to update the LBN mapping, 5 to allocate new blocks on the data and metadata devices, and 1 to commit the superblock. If a transaction is committed only after 1,000 writes, Dmdedup's throughput is 13–34MB/sec—between that of unlimited and single-write transactions. Note that in this case, for all cache sizes over 25%, performance does not depend



(a) Sequential write, 640KB I/O size

(b) Random write, 4KB I/O size

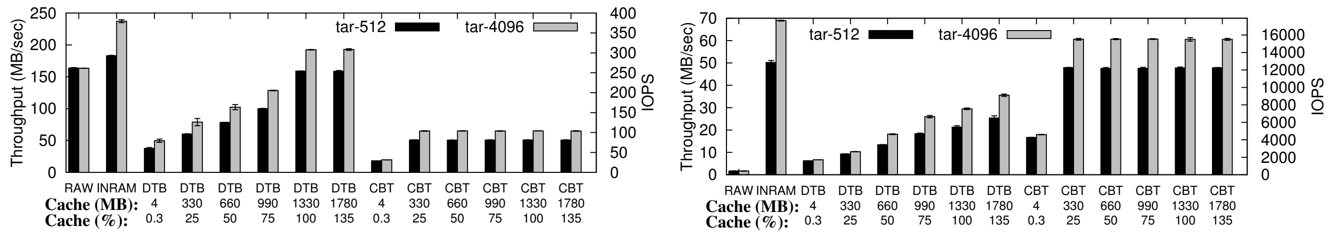
Figure 4: Sequential and random write throughput for the **Unique** dataset. Results are for the raw device and for Dmdedup with different metadata backends and cache sizes. For the CBT backend we varied the transaction size: unlimited, every I/O, and 1,000 writes.



(a) Sequential write, 640KB I/O size

(b) Random write, 4KB I/O size

Figure 5: Sequential and random write throughput for the **All-duplicates** dataset. Results are for the raw device and for Dmdedup with different backends and cache sizes. For the CBT backend, we varied the transaction size: unlimited, every I/O, and 1,000 writes.



(a) Sequential write, 640KB I/O size)

(b) Random write, 4KB I/O size

Figure 6: Sequential and random write throughput for the **Linux-kernels** dataset. Results are for the raw device and for Dmdedup with different metadata backends and cache sizes. For the CBT backend, the transaction size was set to 1,000 writes.

on the cache size but only on the fact that Dmdedup flushes metadata after every 1,000 writes.

For random-write workloads (Figure 4(b)) the raw device achieves 420 IOPS. Dmdedup performs significantly better than the raw device—between 670 and 11,100 IOPS (in 4KB units)—because it makes random writes sequential. Sequential allocation of new blocks is a common strategy in deduplication systems [13,23,27], an aspect that is often overlooked when discussing deduplication's performance impact. We believe that write sequentialization makes primary storage deduplication significantly more practical than commonly perceived.

All-duplicates (Figure 5). This dataset is on the other end of the deduplication ratio range: all writes contain exactly the same data. Thus, after the first write, nothing needs to be written to the data device. As a result, Dmdedup outperforms the raw device by 1.4–2× for the sequential workload in all configurations except CBT with per-write transactions (Figure 5(a)). In the latter case, Dmdedup's throughput falls to 12MB/sec due to the many (ten) metadata writes induced by each user write. Write amplification is lower for All-duplicates than for Unique because the hash index is not updated in the former case. The throughput does not depend on the cache size here because the hash index contains only

Trace	Duration (days)	Written (GB)	Written unique by content (GB)	Written unique by offset (GB)	Read (GB)	Dedup ratio	Dedup block size (B)	Ranges accessed (GB)
Web	21	42.64	22.45	0.95	11.89	2.33	4,096	19.07
Mail	20	1,483.41	110.39	8.28	188.94	10.29	4,096	278.87
Homes	21	65.27	16.83	4.95	15.46	3.43	512	542.32

Table 1: Summary of FIU trace characteristics

one entry and fits even in the 4MB cache. The LBN mapping is accessed sequentially, so a single I/O brings in many cache entries that are immediately reaccessed.

For random workloads (Figure 5(b)), Dmdedup improves performance even further: 2–140× compared to the raw device. In fact, random writes to a disk drive are so slow that deduplicating them boosts overall performance. But unlike the sequential case, the DTB backend with a 4MB cache performs poorly for random writes because LBNs are accessed randomly and 4MB is not enough to hold the entire LBN mapping. For all other cache sizes, the LBN mapping fits in RAM and performance was thus not impacted by the cache size.

The CBT backend caches two copies of the tree in RAM: original and modified. This is the reason why its performance depends on the cache size in the graph.

Linux kernels (Figure 6). This dataset contains the source code of 40 Linux kernels from version 2.6.0 to 2.6.39, archived in a single tarball. We first used an unmodified `tar`, which aligns files on 512B boundaries (*tar-512*). In this case, the tarball size was 11GB and the deduplication ratio was 1.18. We then modified `tar` to align files on 4KB boundaries (*tar-4096*). In this case, the tarball size was 16GB and the deduplication ratio was 1.88. Dmdedup uses 4KB chunking, which is why aligning files on 4KB boundaries increases the deduplication ratio. One can see that although tar-4096 produces a larger logical tarball, its physical size ($16\text{GB}/1.88 = 8.5\text{GB}$) is actually smaller than the tarball produced by tar-512 ($11\text{GB}/1.18 = 9.3\text{GB}$).

For sequential writes (Figure 6(a)), the INRAM backend outperforms the raw device by 11% and 45% for tar-512 and tar-4096, respectively. This demonstrates that storing data in a deduplication-friendly format (tar-4096) benefits performance in addition to reducing storage requirements. This observation remains true for other backends. (For CBT we show results for the 1000-write transaction configuration.) Note that for random writes (Figure 6(b)), the CBT backend outperforms DTB. Unlike a hash table, the B-tree scales well with the size of the dataset. As a result, for both 11GB and 16GB tarballs, B-trees fit in RAM, while the on-disk hash table is accessed randomly and cannot be cached as efficiently.

4.2.2 Trace Replay

To evaluate Dmdedup's performance under realistic workloads, we used three production traces from Florida International University (FIU): Web, Mail, and Homes [1, 10]. Each trace was collected in a significantly different environment. The Web trace originates from two departments' Web servers; Homes from a file server that stores the home directories of a small research group; and Mail from a departmental mail server. Table 1 presents relevant characteristics of these traces.

FIU's traces contain data hashes for every request. We applied a patch from Koller and Rangaswami [10] to Linux's *btreplay* utility so that it generates unique write content corresponding to the hashes in the traces, and used that version to drive our experiments. Some of the reads in the traces access LBNs that were not written during the tracing period. When serving such reads, Dmdedup would normally generate zeroes without performing any I/O; to ensure fairness of comparison to the raw device we pre-populated those LBNs with appropriate data so that I/Os happen for every read.

The Mail and Homes traces access offsets larger than our data device's size (146GB). Because of the indirection inherent to deduplication, Dmdedup can support a logical size larger than the physical device, as long as the deduplication ratio is high enough. But replaying the trace against a raw device (for comparison) is not possible. Therefore, we created a small device-mapper target that maintains an LBN$\rightarrow$PBN mapping in RAM and allocates blocks sequentially, the same as Dmdedup. We set the sizes of both targets to the maximum offset accessed in the trace. Sequential allocation favors the

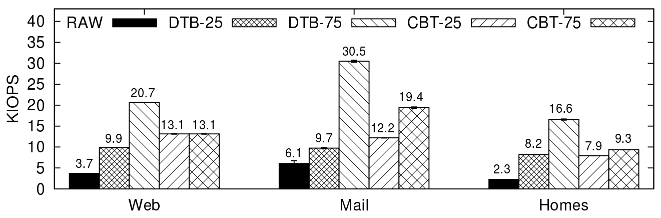


Figure 7: Raw device and Dmdedup throughput for FIU's Web, Mail, and Homes traces. The CBT backend was setup with 1000-writes transaction sizes.

raw device, but even with this optimization, Dmdedup significantly outperforms the raw device.

We replayed all traces with unlimited acceleration. Figure 7 presents the results. Dmdedup performs 1.6–7.2× better than a raw device due to the high redundancy in the traces. Note that write sequentialization can harm read performance by randomizing the read layout. Even so, in the FIU traces (as in many real workloads) the number of writes is higher than the number of reads due to large file system buffer caches. As a result, Dmdedup's overall performance remains high.

4.2.3 Performance Comparison to Lessfs

Dmdedup is a practical solution for real storage needs. To our knowledge there are only three other deduplication systems that are free, open-source, and widely used: Lessfs [11], SDFS [20], and ZFS [4]. We omit research prototypes from this list because they are usually unsuitable for production. Both Lessfs and SDFS are implemented in user space, and their designs have much in common. SDFS is implemented using Java, which can add high overhead. Lessfs and Dmdedup are implemented in C, so their performance is more comparable.

Table 2 presents the time needed to extract a tarball of 40 Linux kernels (uncompressed) to a newly created file system. We experimented with plain Ext4, Dmdedup with Ext4 on top, and Lessfs deployed above Ext4. When setting up Lessfs, we followed the best practices described in its documentation. We experimented with BerkleyDB and HamsterDB backends with transactions on and off, and used 4KB and 128KB chunk sizes. The deduplication ratio was 2.4–2.7 in these configurations. We set the Lessfs and Dmdedup cache sizes to 150MB, which was calculated for our dataset using the db_stat tool from Lessfs. We configured Dmdedup to use the CBT backend because it guarantees transactionality, similar to the databases used as Lessfs backends.

	Ext4	**Dm dedup 4KB**	**Lessfs**		
			BDB 4KB	**BDB 128KB**	**HamsterDB 128KB TransOFF**
Time (sec)	649	521	1,825	1,413	814

Table 2: Time to extract 40 Linux kernels on Ext4, Dmdedup, and Lessfs with different backends and chunk sizes. We turned transactions off in HamsterDB for better performance.

Dmdedup improves plain Ext4 performance by 20% because it eliminates duplicates. Lessfs with the BDB backend and a 4KB chunk size performs 3.5× slower than Dmdedup. Increasing the chunk size to 128KB improves Lessfs's performance, but it is still 2.7× slower than Dmdedup with 4KB chunks. We achieved the highest Lessfs performance when using the HamsterDB backend with 128KB and disabling transactions. However, in this case we sacrificed both deduplication ratio and transactionality. Even then, Dmdedup performs 1.6× faster than Lessfs while providing transactionality and a high deduplication ratio. The main reason for poor Lessfs performance is its high CPU utilization—about 87% during the run. This is caused by FUSE, which adds significant overhead and causes many context switches [19]. To conclude, Dmdedup performs significantly better than other popular, open-source solutions from the same functionality class.

Unlike Dmdedup and Lessfs, ZFS falls into a different class of products because it does not add deduplication to existing file systems. In addition, ZFS deduplication logic was designed to work efficiently when all deduplication metadata fits in the cache. When we limited the ZFS cache size to 1GB, it took over two hours for `tar` to extract the tarball. However, when we made the cache size unlimited, ZFS was almost twice as fast as Dmdedup+Ext4. Because of its complexity, ZFS is hard to set up in a way that provides a fair comparison to Dmdedup; we plan to explore this in the future.

5 Related Work

Many previous deduplication studies have focused on backup workloads [18, 27]. Although Dmdedup can be used for backups, it is intended as a general-purpose deduplication system. Thus, in this section we focus on primary-storage deduplication.

Several studies have incorporated deduplication into existing file systems [15, 21] and a few have designed deduplicating file systems from scratch [4]. Although there are advantages to deduplicating inside a file system, doing so is less flexible for users and researchers because they are limited to that system's architecture. Dmdedup does not have this limitation; it can be used with any file system. FUSE-based deduplication file systems are another popular design option [11,20]; however, FUSE's high overhead makes this approach impractical for production environments [19]. To address performance problem, El-Shimi et al. built an in-kernel deduplication file system as a filter driver for Windows [8] at the price of extra development complexity.

The systems most similar to ours are Dedupv1 [13] and DBLK [23], both of which deduplicate at the block level. Each is implemented as a user-space iSCSI target, so their performance suffers from additional data copies and context switches. DBLK is not publicly available.

It is often difficult to experiment with existing research systems. Many are raw prototypes that are unsuitable for extended evaluations, and only a few have made their source code available [4,13]. Others were intended for specific experiments and lack experimental flexibility [24]. High-quality production systems have been developed by industry [3, 21] but it is hard to compare against unpublished, proprietary industry products. In contrast, Dmdedup has been designed for experimentation, including a modular design that makes it easy to try out different backends.

6 Conclusions and Future Work

Primary-storage deduplication is a rapidly developing field. To test new ideas, previous researchers had to either build their own deduplication systems or use closed-source ones, which hampered progress and made it difficult to fairly compare deduplication techniques. We have designed and implemented Dmdedup, a versatile and practical deduplication block device that can be used by regular users and researchers. We developed an efficient API between Dmdedup and its metadata backends to allow exploration of different metadata management policies. We designed and implemented three backends (INRAM, DTB, and CBT) for this paper and evaluated their performance and resource use. Extensive testing demonstrates that Dmdedup is stable and functional, making evaluations using Dmdedup more realistic than experiments with simpler prototypes. Thanks to reduced I/O loads, Dmdedup improves system throughput by 1.5–6$\times$ under many realistic workloads.

Future work. Dmdedup provides a sound basis for rapid development of deduplication techniques. By publicly releasing our code we hope to spur further research in the systems community. We plan to develop further metadata backends and cross-compare them with each other. For example, we are considering creating a log-structured backend.

Compression and variable-length chunking can improve overall space savings but require more complex data management, which might decrease system performance. Therefore, one might explore a combination of on-line and off-line deduplication.

Aggressive deduplication might reduce reliability in some circumstances; for example, if all copies of an FFS-like file system's super-block were deduplicated into a single one. We plan to explore techniques to adapt the level of deduplication based on the data's importance. We also plan to work on automatic scaling of the hash index and LBN mapping relative to the size of metadata and data devices.

Acknowledgments. This work was made possible in part thanks to NSF awards CNS-1305360, IIS-1251137, and CNS-1302246, as well as an EMC Faculty award. We also thank Teo Asinari, Mandar Joshi, Atul Karmarkar, Gary Lent, Amar Mudrankit, Eric Mueller, Meg O'Keefe, and Ujwala Tulshigiri for their valuable contributions to this projects. We appreciate Raju Rangaswami's and Ricardo Koller's help on understanding and replaying FIU traces.

References

[1] Storage Networking Industry Association. IOTTA Trace Repository. http://iotta.snia.org, accessed in May 2014.

[2] R. E. Bohn and J. E. Short. How Much Information? 2009 Report on American Consumers, December 2009. http://hmi.ucsd.edu/pdf/HMI_2009_ConsumerReport_Dec9_2009.pdf, accessed in May 2014.

[3] W. Bolosky, S. Corbin, D. Goebel, and J. Douceur. Single Instance Storage in Windows 2000. In *Proceedings of the ATC Conference*, 2000.

[4] J. Bonwick. ZFS Deduplication, November 2009. http://blogs.oracle.com/bonwick/entry/zfs_dedup, accessed in May 2014.

[5] Y. Chen, K. Srinivasan, G. Goodson, and R. Katz. Design Implications for Enterprise Storage Systems via Multi-Dimensional Trace Analysis. In *Proceedings of the SOSP Symposium*, 2011.

[6] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[7] M. Dutch. Understanding Data Deduplication Ratios. Technical report, SNIA Data Management Forum, 2008.

[8] A. El-Shimi, R. Kalach, A. Kumar, A. Oltean, J. Li, and S. Sengupta. Primary Data Deduplication—Large Scale Study and System Design. In *Proceedings of the ATC Conference*, 2012.

[9] J. Gray and C. V. Ingen. Empirical Measurements of Disk Failure Rates and Error Rates. Technical Report MSR-TR-2005-166, Microsoft Research, December 2005.

[10] R. Koller and R. Rangaswami. I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance. In *Proceedings of the FAST Conference*, 2010.

[11] Lessfs, January. www.lessfs.com, accessed in May 2014.

[12] M. Lu, D. Chambliss, J. Glider, and C. Constantinescu. Insights for Data Reduction in Primary Storage: A Practical Analysis. In *Proceedings of the SYSTOR Conference*, 2012.

[13] D. Meister and A. Brinkmann. dedupv1: Improving Deduplication Throughput using Solid State Drives (SSD). In *Proceedings of the MSST Conference*, 2010.

[14] D. Meyer and W. Bolosky. A Study of Practical Deduplication. In *Proceedings of the FAST Conference*, 2011.

[15] A. More, Z. Shaikh, and V. Salve. DEXT3: Block Level Inline Deduplication for EXT3 File System. In *Proceedings of the OLS*, 2012.

[16] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the SOSP Symposium*, 2001.

[17] P. Nath, M. Kozuch, D. O'Hallaron, J. Harkes, M. Satyanarayanan, N. Tolia, and M. Toups. Design Tradeoffs in Applying Content Addressable Storage to Enterprise-scale Systems Based on Virtual Machines. In *Proceedings of the ATC Conference*, 2006.

[18] S. Quinlan and S. Dorward. Venti: a New Approach to Archival Storage. In *Proceedings of the FAST Conference*, 2002.

[19] A. Rajgarhia and A. Gehani. Performance and Extension of User Space File Systems. In *Proceedinsgs of the Symposium On Applied Computing*. ACM, 2010.

[20] Opendedup - SDFS. www.opendedup.org, accessed in May 2014.

[21] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the FAST Conference*, 2012.

[22] J. Thornber. *Persistent-data library*. https://git.kernel.org/cgit/linux/kernel/git/torvalds/linux.git/tree/Documentation/device-mapper/persistent-data.txt, accessed in May 2014.

[23] Y. Tsuchiya and T. Watanabe. DBLK: Deduplication for Primary Block Storage. In *Proceedings of the MSST Conference*, 2011.

[24] A. Wildani, E. Miller, and O. Rodeh. HANDS: A Heuristically Arranged Non-Backup In-line Deduplication System. In *Proceedings of the ICDE Conference*, 2013.

[25] F. Xie, M. Condict, and S. Shete. Estimating Duplication by Content-based Sampling. In *Proceedings of the ATC Conference*, 2013.

[26] E. Zadok. Writing Stackable File Systems. *Linux Journal*, 05(109):22–25, May 2003.

[27] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the FAST Conference*, 2008.

SkyPat: C++ Performance Analysis and Testing Framework

Ping-Hao Chang, Kuan-Hung Kuo, Der-Yu Tsai, Kevin Chen, Luba Tang
Skymizer Software
{peter,ggm,a127a127,kevin,luba}@skymizer.com

Abstract

This paper introduces SkyPat, a C++ performance analysis toolkit on Linux. SkyPat combines unit tests and `perf_event` to give programmers the power of white-box performance analysis.

Unlike traditional tools that manipulate entire program as a black-box, SkyPat works on a region of code like a white-box. It is used as a normal unit test library. It provides macros and assertions, into which `perf_events` are embedded, to ensure correctness and to evaluate performance of a region of code. With `perf_events`, SkyPat can evaluate running time precisely without interference to scheduler. Moreover, `perf_event` also gives SkyPat accurate cycle counts that are useful for tools that are sensitive to variance of timing, such as compilers.

We develop SkyPat under the new BSD license, and it is also the unit-test library of the "bold" project.

1 Introduction

SkyPat is developed by a group of the compiler developers to satisfy compiler developers' needs. From compiler developers' view, correctness and performance evaluation are grand challenges for engineering. Compiler optimizations have no guarantee of performance improvement. Sometimes optimizations degrade performance, and sometimes they introduce new faults in a program. Compiler developers need a tool to verify correctness and performance of each compiler optimizations at the same time.

Traditional tools that evaluate the whole program do not fit our demands. Compilers perform optimization based on knowledge it has to a region of code, such as loops or data flows. We need libraries that can evaluate only a region of code that optimization is interesting in.

For compiler developers, integrating unit-test and performance evaluation libraries for a piece of code is very rational. SkyPat is such library: by using SkyPat, users can evaluate correctness and performance by writing test-cases to evaluate a region of code.

Usually, unit-test tools and performance evaluation tools are separated tools. For example, *GoogleTest* [2] is well-known C++ unit-test framework. *GoogleTest* can evaluate correctness but cannot evaluate performance. Besides, *perf* [1] is well-known performance evaluation toolkit in Linux. *perf* can evaluate performance of programs, including its running time, cycles and so on. Although *perf* can evaluate whole program, using *perf* to evaluate a region of code is difficult.

In this paper, we introduce SkyPat, which combines unit-test and performance evaluation. Programmers only need to write and execute unit-tests and they can get correctness and performance. With the help of `perf_event` of Linux kernel, SkyPat can provide precise timer and additional runtime information to measure a region of code. By integrating unit-test and performance evaluation, SkyPat make evaluation of a region of code easier.

The organization of this paper is as follows. Related work is in Section 2. We present SkyPat's design and implementation in Section 3 and shows SkyPat testing and performance framework in Section 4. At last, we conclude this paper in Section 5.

2 Related Work

Oprofile[3] and *perf* are two most popular performance evaluation tools. Oprofile is capable to evaluate not only the whole system but also a single program. Before versions 0.9.7 and earlier, it was based on sampling-based daemon that collects runtime information. Because sampling-based daemon wasted system resources, Linux community creates new interfaces, called "Performance Counter"[4] or "`perf_event`". After that, Oprofile is "`perf_event`"-based in the later version.

By the success in "Performance Counter", Linux community builds another performance evaluation tool, called *perf*, based on "Performance Counter", too. *Perf* is a performance evaluation toolkit with no daemon to collect runtime information. *perf* gets runtime information by kernel directly rather than collecting by daemon, therefore, *perf* eliminates lots of overhead to bookkeep profiling information and becomes faster.

Both OProfile and *perf* evaluate performance of the entire program, not a region of code. And of course, it makes some efforts to integrate them with unit-test frameworks.

Regarding unit-test frameworks, *GoogleTest* has become popular and has been adapted by many projects recently. *GoogleTest* is a xUnit test framework for C++ programs. By providing ASSERT and EXPECT macros, *GoogleTest* helps programmers to verify program's correctness by writing test-cases. While executing test-cases, a program stops immediately if it meets a fatal error. If a program meets a non-fatal error, the program shows the runtime value and expected value on the screen.

3 Implementation and Design

SkyPat is a C++ performance analyzing and testing framework on Linux platforms. We refer to the concept of *GoogleTest* and extend its scope from testing framework into Performance Analysis Framework. With the help of SkyPat, programmers who wants to analyze their program, only need to write test cases, and SkyPat tests programs' correctness and performance just like normal unit-test frameworks.

SkyPat provides ASSERT/EXPECT macros for correctness checking and PERFORM macro for performance evaluation. ASSERT is assertion for fatal condition testing and EXPECT is non-fatal assertion. That is to say, if a condition of ASSERT fails, the test fails and stops immediately. On the other hand, when the condition of EXPECT fails, it just shows messages on screen to indicate that is a non-fatal failure and the test keeps going on.

A PERFORM macro is used to arbitrarily wrap an block of code in a testee function. It invokes `perf_events` at the beginning and the end of the block of code to measure the performance. When a program executes at the beginning of the region of code, the PERFORM macro calls a system call to kernel to register a performance monitor to gather process runtime information, such as execution time. When program executes in the end of the region of code, a system call is sent to kernel automatically to disable the monitor. SkyPat calculates the difference of time between the beginning and the end to get the period of runtime information of the region of code.

4 SkyPat Testing and Performance Framework

Here are some examples to show how to use SkyPat to evaluate correctness and performance.

4.1 Declare Test Cases and Test Functions

To create a test, users use the PAT_F() macro to define a test function. A test function can be thought as a ordinary C function without return value. Several similar test functions for the same input data can be grouped as a test case.

Figure 1 shows how to define a test-case and test-functions.

Every PAT_F macro declares a test, with two parameters: test-case and test-function names. In Figure 1, "AircraftCase" is the name of test-case. "take_off_test" and "landing_test" are the names of test-function belongs to "AircraftCase" test case. Test functions grouped in the same test-case are meant to be logically related. Users put ASSERT/EXPECT and PERFORM macros in a test function to evaluate correctness and performance. These macros is described in the following section.

```
PAT_F(AircraftCase, take_off_test)
{
   // Test Code
}

PAT_F(AircraftCase, landing_test)
{
   // Test Code
}
```

Figure 1: Example for declaring a test

```
PAT_F(MyCase, fibonacci_test)
{
  ASSERT_TRUE(0 != fibonacci(10));
  EXPECT_EQ(fibonacci(10), 2);
  ASSERT_NE(fibonacci(10), 3);
}

PAT_F(MyCase, AP_test)
{
  ...
```

Figure 2: Example of assertions

```
[ RUN      ] MyCase.fibonacci_test
[  FAILED  ]
main.cpp:53: error: failed to expect
Value of: 2 == fibonacci(10)
  Actual:   false
  Expected: true
[ RUN      ] MyCase.AP_test
[       OK ]
```

Figure 3: Output of Figure 2

4.2 Correctness Checking

We copy the concept from *GoogleTest* for our correctness evaluation.

There are several variants of ASSERT macros, ASSERT_TRUE/FALSE, ASSERT_EQ/NE (equal) and ASSERT_GT/GE/LT/TE (great/less) series. If the condition of ASSERT fails, the test will stop and exit the test immediately. EXPECT macros, like ASSERT macros, there are also some similar variants. If the condition of EXPECT fails, the test will not stop but keep execute and display the expected result and real result on screen.

Figure 2 shows how fatal and non-fatal assertions work. "fibonacci" is a testee for illustration. There are three assertions: two fatal and one non-fatal.

Figure 3 shows the output of Figure 2. As we mentioned before, ASSERT assertions stop the execution immediately and EXPECT assertions try to keep the execution going on.

4.3 Performance Evaluation

A PERFORM macro measures the performance of a block of code which it wraps up. Figure 4 shows how to use PERFORM macro.

```
PAT_F(MyCase, fibonacci_perf_test)
{
  PERFORM {
    fibonacci(40);
  }
}
```

Figure 4: Example of PERFORM

```
[ RUN      ] MyCase.fibonacci_test
[CXT SWITCH]       3
[ TIME (ns)]  2363214415
```

Figure 5: Output of Figure 4

The PERFORM macro registers a performance monitor at the beginning of the code which its wraps up and detaches the monitor when leaving the block of code. SkyPat calculates and remembers the performance details whenever detaching a performance monitor.

The result is shown in Figure 5. SkyPat measures the execution time and the number of context-switches during the region of code. It saves efforts at complicated interaction between *perf* and the region of code. Users just use macros, just works like writing a test program, and they can easily get the runtime information of the region of code.

For now, SkyPat measures few information such as the run-time clock cycles. We will add more features, such as cache miss and page faults, in the near future.

4.4 Run All Test Cases

To integrate all test-case, user should call *Initialize* and *RunAll* in their program. `Initialize(&argc, argv)` initializes outputs. `RunAll()` runs all the tests you've declared and prints the results on screen. If any test fails, then `RunAll()` returns non-zero value.

```
int main(int argc, char* argv[])
{
  pat::Test::Initialize(&argc, argv);
  pat::Test::RunAll();
}
```

Figure 6: Example of Initialize and RunAll

5 Conclusion and Future Works

By integrating unit-test framework and performance evaluation tool, user can get correctness and performance metrics for a region of code by writing test-cases. Users can get the performance between the region of code defined by themselves. For programs which needs high precise timing information and other runtime information of the region of code, such as compiler, SkyPat can give them more ability to measure the bottleneck of regions of a program.

References

[1] Arnaldo Carvalho de Melo, Redhat, "The New Linux 'perf' tools" in *17 International Linux System Technology Conference (Linux Kongress), 2010*

[2] GoogleTest, Google, https://code.google.com/p/googletest/

[3] OProfile, http://oprofile.sourceforge.net/about/

[4] Ingo Molnar, "Performance Counters for Linux". *Linux Weekly News, 2009*. http://lwn.net/Articles/337493/

Computationally Efficient Multiplexing of Events on Hardware Counters

Robert V. Lim
University of California, Irvine
roblim1@ics.uci.edu

David Carrillo-Cisneros
University of California, Irvine
dcarril@ics.uci.edu

Wail Y. Alkowaileet
University of California, Irvine
walkowai@ics.uci.edu

Isaac D. Scherson
University of California, Irvine
isaac@ics.uci.edu

Abstract

This paper proposes a novel approach for scheduling n performance monitoring events onto m hardware performance counters, where $n > m$. Whereas existing scheduling approaches overlook monitored task information, the proposed algorithm utilizes the monitored task's behavior and schedules the combination of the most costly events. The proposed algorithm was implemented in Linux Perf Event subsystem in kernel space (build 3.11.3), which provides finer granularity and less system perturbation in event monitoring when compared to existing user space approaches. Benchmark experiments in PARSEC and SPLASH.2x suites compared the existing round-robin scheme with the proposed rate-of-change approach. Results demonstrate that the rate-of-change approach reduces the mean-squared error on average by 22%, confirming that the proposed methodology not only improves the accuracy of performance measurements read, but also makes scheduling multiple event measurements feasible with a limited number of hardware performance counters.

1 Introduction

Modern performance tools (PAPI, Perf Event, Intel vTune) incorporate hardware performance counters in systems monitoring by sampling low-level hardware events, where each performance monitoring counter (PMC) is programmed to count the number of occurrences of a particular event, and its counts are periodically read from these registers. The monitored results collectively can provide insights into how the task behaves on a particular architecture. Projecting performance metrics such as instructions-per-cycle (IPC), branch mispredictions, and cache utilization rates not only helps analysts identify hotspots, but can lead to code optimization opportunities and performance tuning enhancements. Hardware manufacturers provide hundreds of performance events that can be programmed onto the PMCs for monitoring. For instance, Intel provides close to 200 events for the current i7 architecture [6], while AMD provides close to 100 events [12]. Other architectures that provide event monitoring capabilities include NVIDIA's `nvprof` and Qualcomm's Adereno profilers [11, 14]. While manufacturers have provided an exhaustive list of event types to monitor, the issue is that microprocessors usually provide two to six performance counters for a given architecture, which restricts the number of events that can be monitored simultaneously.

Calculating performance metrics involves n low-level hardware events, and modern microprocessors provide m physical counters (two to six), making scheduling multiple performance events impractical when $n > m$. A single counter can monitor only one event at a time, which means that two or more events assigned to the same register cannot be counted simultaneously (conflicting events) [9].

Monitoring more events than available counters can be achieved with time interpolation techniques, such as multiplexing and trace alignment. Multiplexing consists of scheduling events for a fraction of the execution and extrapolating the full behavior of each metric from its samples. Trace alignment, on the other hand, involves collecting separate traces for each event run and combining the independent runs into a single trace.

Current approximation techniques for reconstructing event traces yield estimation errors, which provides inaccurate measurements for performance analysts [10]. The estimation error increases with multiplexing be-

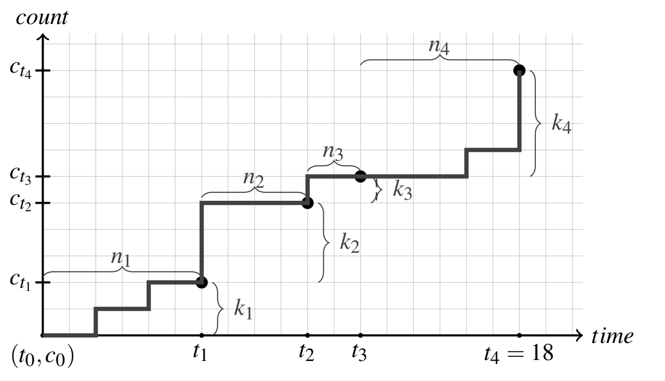


Figure 1: An example of a sequence of sampled values. The sampling times $t_1 \ldots t_4$ and counts $c_{t_0} = 0, \ldots c_{t_4}$ are known.

cause each event timeshares the PMC with the other events, which results in loss of information when the event is not being monitored at a sampled interval. Trace alignment may not be feasible in certain situations, where taking multiple runs of the same application for performance monitoring might take days or weeks to complete. In addition, the authors have shown that between-runs variability affects the correlation between the sampled counts for monitored events, due to hardware interrupts, cache contention, and system calls [16]. Current implementations schedule monitoring events in a round-robin fashion, ignoring any information about the program task. Opportunities for better event scheduling exist if information about the behavior of the task is taken into account, an area we address in this paper.

This paper is organized as follows. Section 2 discusses previous work. Our multiplexing methodology is presented in Section 3. Section 4 evaluates the experimental results. Lastly, Section 5 concludes with future work.

2 Previous Work

To the best of our knowledge, there has not been any prior work similar to our methodology for multiplexing *n* performance events onto *m* hardware counters. The next subsections discuss several performance monitoring tools and its respective multiplexing strategy.

2.1 Performance Monitoring Tools

Performance monitoring tools provide access to hardware performance counters either through user space or kernel space.

Performance Application Programming Interface (PAPI) is an architecture independent framework that provides access to generalized high-level hardware events for modern processors, and low-level native events for a specific processor [2]. PAPI incorporates MPX and a high resolution interval timer to perform counter multiplexing [9]. The TAU Performance System, which integrates PAPI, is a probed-based instrumentation framework that profiles applications, libraries, and system codes, where execution of probes become part of the normal control flow of the program [15]. PAPI's ease of use, and feature-rich capabilities make the framework a top choice in systems running UNIX/Linux, ranging from traditional microprocessors to high-performance heterogeneous architectures.

Perfmon2, a generic kernel-level performance monitoring interface, provides access to the hardware performance monitoring unit (PMU) and supports a variety of architectures, including Cray X2, Intel, and IBM PowerPC [4]. Working at the kernel level provides fine granularity and less system perturbation when accessing hardware performance counters, compared to user space access [17]. Scheduling multiple events in Perfmon2 is handled via round-robin, where the order of event declaration determines its initial position in the queue. Linux Perf Event subsystem is a kernel level monitoring platform that also provides multi-architectural support (x86, PowerPC, ARM, etc.) [13]. Perf has been mainlined in the Linux kernel, making Perf monitoring tool available in all Linux distributions. Our proposed methodology was implemented in Perf Event.

2.2 Perf Event in Linux

Perf Event samples monitoring events asynchronously, where users set a *period* (at every i^{th} interval) or a *frequency* (the number of occurrence of events). Users declare an event to monitor by creating a file descriptor, which provides access to the performance monitoring unit (PMU). The PMU state is loaded onto the counter register with a `perf_install_in_context` call. Similar to Perfmon2, the current criteria for multiplexing events is round-robin.

A monitoring Perf Event can be affected under the following three scenarios: `hrtimer`, scheduler tick, and interrupt context.

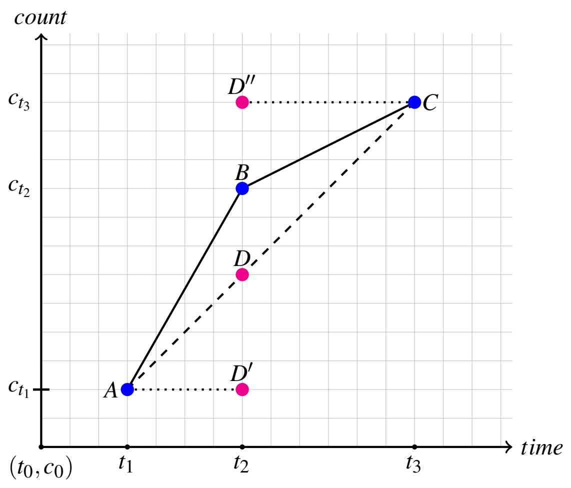


Figure 2: Triangle representing rate-of-change calculation for the recent three observations A, B, and C.

2.2.1 `hrtimer`

`hrtimer` [5] is a high resolution timer that gets triggered when the PMU is overcommitted [7]. `hrtimer` invokes `rotate_ctx`, which performs the actual multiplexing of events on the hardware performance counters and is where our rate-of-change algorithm is implemented.

2.2.2 Scheduler tick

Performance monitoring events and its count values are removed and reprogrammed on the PMU registers during each operating system scheduler tick, usually set at *HZ* times per second.

2.2.3 Interrupt context

A non-maskable interrupt (NMI) triggers a PMU interrupt handler during a hardware overflow, usually when a period declared by the user has been reached. `perf_rotate_context` completes the interrupt context by multiplexing the events on the PMC.

Our methodology uses `hrtimer` to perform time-division multiplexing. That is, at each `hrtimer` triggered, Perf Event timeshares the events with the performance counters. During the intervals that the events are not monitored, the events are linearly interpolated to estimate the counts [9].

2.3 Linear interpolation

To define linear interpolation for asynchronous event sampling, we will first define a sample, and then use a pair of samples to construct a linear interpolation.

A sample $s_i = (t_i,\ c_{t_i})$ is the i-th sample of a PMC counting the occurrences of an arbitrary event. The sample s_i occurs at time t_i and has a value c_{t_i}. We define:

$$k_i = c_{t_i} - c_{t_{i-1}} \tag{1}$$

$$n_i = t_i - t_{i-1} \tag{2}$$

as the increments between samples s_{i-1} and s_i for an event's count and time, respectively.

The slope of the linear interpolation between the two samples is defined as follows:

$$m_i = \frac{k_i}{n_i} \tag{3}$$

Since all performance counters store non-negative integers, then $0 \le k_i, 0 \le n_i, 0 \le m_i$, for all i. An event sample represents a point in an integer lattice. Figure 1 displays sampled values and the variables defined above.

3 Multiplexing Methodology

Time interpolation techniques for performance monitoring events have shown large accuracy errors when reconstructing event traces [10]. Although increasing the number of observations correlates with more accurate event traces, taking too many samples may adversely affect the quality of the monitored behavior, since each sample involves perturbing the system. Linear interpolation techniques has demonstrated its effectiveness in reconstructing unobserved event traces [8, 9].

Our rate-of-change algorithm increases the amount of monitoring time for events that do not behave linearly. Our intuition tells us that higher errors will occur for non-linear behaved events when reconstructing event traces with linear interpolation techniques. The current round-robin scheme does not detect varying behavior since it schedules each event indiscriminately, missing the opportunity to reduce scheduling time for events where linear interpolation may have been sufficient for reconstruction.

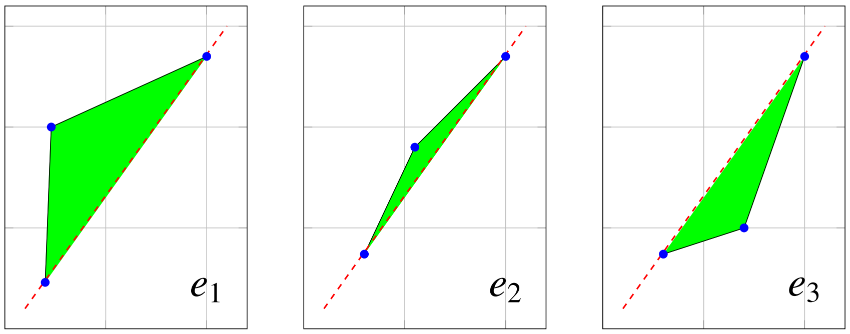


Figure 3: Triangle cost function for events e_1, e_2, and e_3

3.1 Rate-of-Change Definition

To motivate our proposed scheduling algorithm, we define rate-of-change as follows. Let $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$ be the last three observations for a given event in a current program run, where x, y represent the observed time and count, respectively (Sec. 2.3). A triangle can be constructed in an integer lattice using the three observations, where the triangle's area represents the loss of information if $B_{x,y}$ were skipped (Fig. 2). Based on the principle of locality in software applications [3], we hypothesize that the loss of information due to an unknown $B_{x,y}$ is similar to the potential loss of information due to skipped future observations. In Figure 2, the dashed line represents an extracted trace from a linear interpolator, provided that $A_{x,y}$ and $C_{x,y}$ are known, and serves as our scheduling criteria.

3.2 Triangle Cost Function

The triangle cost function is calculated as follows. For any three observations $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$, we have

$$\begin{array}{lll} D_x = B_x & D_y = \delta_y + A_y & \\ D'_x = B_x & D'_y = A_y & \delta_y = \dfrac{C_y - A_y}{C_x - A_x} \cdot (B_x - A_x) \\ D''_x = B_x & D''_y = C_y & \end{array}$$

Definition 1 *The rate-of-change in observations $A_{x,y}$, $B_{x,y}$, and $C_{x,y}$ is given by the sum of the areas $\triangle_{ABD}$ and $\triangle_{BCD}$.*

$\triangle_{ABD}$ is calculated as follows:

$$\triangle_{ABD} = \frac{B_x - A_x}{2} \cdot (B_y - A_y - \delta_y) \quad (4)$$

The scheduling cost, C_{ABD}, is determined as follows:

$$C_{ABD} = \left| \frac{B_y - A_y - \delta_y}{2} \right| \cdot \delta_t \quad (5)$$

Calculations for $\triangle_{BCD}$ and C_{BCD} are similar to (4) and (5), respectively:

$$\triangle_{BCD} = \frac{C_x - B_x}{2} \cdot (B_y - A_y - \delta_y) \quad (6)$$

$$C_{BCD} = \left| \frac{B_y - A_y - \delta_y}{2} \right| \cdot \delta_t \quad (7)$$

3.3 Rate-of-Change as Scheduling Criteria

The rate-of-change algorithm calculates a cost function based on the recent event observations to determine whether the monitoring event should be scheduled next. A smaller triangle area implies less variability, since the area will be equivalent to a linear slope rate, and is easier to extrapolate. Conversely, a greater triangle area reflects sudden changes in the observations, or higher variability, which means that those events should be prioritized over others.

Figure 3 illustrates how the rate-of-change algorithm calculates the scheduling cost function for each event e_1, e_2, and e_3 with respect to the linear interpolator (red dashed line). The triangle area shaded in green represents the cost of scheduling a particular e_i. Event e_2 exhibits nearest linear behavior, which will be placed in the rear of the scheduling queue. Note that the constructed triangle for e_3 reflects with the linear interpolator, which is addressed with the absolute value sign in Equation 5. The objective of our rate-of-change algorithm is to "punish" non-linearly behaved events by scheduling those events ahead of the queue. After running our algorithm, the scheduling queue will be arranged as follows: $\{e_1, e_3, e_2\}$.

Hardware		Cache			
Event	Period	Event	Period	Event	Period
instructions	4,000,000	L1-dCache-hits	1,000,000	dTLB-Cache-hits	1,100,000
cache-references	9,500	L1-dCache-misses	7,000	dTLB-Cache-misses	750
cache-misses	150	L1-dCache-prefetch	7,000	iTLB-Cache-hits	4,200,000
branch-instructions	850,000	L1-iCache-hits	1,800,000	iTLB-Cache-misses	500
branch-misses	10,000	L1-iCache-misses	50,000	bpu-Cache-hits	900,000
stalled-cycles (frontend)	15,000	LL-Cache-hits	3,100	bpu-Cache-misses	600,000
stalled-cycles (backend)	1,500,000	LL-Cache-misses	75	node-Cache-hits	70
ref-cpu-cycles	2,000,000	LL-Cache-prefetch	500	node-Cache-misses	70
				node-Cache-prefetch	50

Table 1: Generalized Perf Events and its period settings.

Event starvation prevented

Our rate-of-change scheduling criteria prevents event starvation because the cost of scheduling increases as time since the last scheduled event (δ_t) increases; hence, preventing any events from not being scheduled (Eq. 5).

Computationally efficient

Our rate-of-change algorithm is computationally efficient because for every `hrtimer` triggered, only two integer multiplications and two integer divisions are taking place. Below is a snippet of `C` code for our computation:

```
delta_y = ((Cy-Ay)/(Cx-Ax)) * (Bx-Ax);
cost = ((By-Ay-delta_y) / 2) * d_time;
```

In cases where $C_x - A_x = 0$, δ_y is set to 0 which implies $C_x = A_x$, or that no changes have occurred since observation A_x.

4 Experiments and Results

4.1 Experiments

In order to verify our proposed rate-of-change methodology, we ran a subset of PARSEC benchmarks [1] and SPLASH.2X benchmarks [18], listed in Table 2, while sampling the events from the PMC periodically. SPLASH.2X is the SPLASH-2 benchmark suite with bigger inputs drawn from actual production workloads.

Our goal was to make use of `hrtimer` in our multiplexing methodology, which was released in kernel version `3.11.rc3` [7].

The generalized Perf Events for this experiment, listed in Table 1, were periodically sampled for each benchmark run. Each of the periods was determined by trial-and-error using the following formula.

$$samples/msec = \frac{\mu_{nr.samples}}{\mu_{t.elapsed}} \qquad (8)$$

Ideal samples per milliseconds is 1, where the number of samples taken for an event is proportional to the time spent monitoring the event.

To account for between-runs variability, each of the events listed in Table 1 executed ten times for each benchmark package with `simlarge` as input, which ran on average 15 seconds each. Each execution consisted of programming one single event in the PMC to disable multiplexing, which serves as the baseline comparison. In addition, the events were multiplexed with the performance counters in the same run under the current round-robin scheme and with our rate-of-change methodology. Each of the ten runs was sorted in ascending order, based on counts for each scheduling strategy, and the fourth lowest value was selected as our experimental result for the single event baseline comparison, and for the two multiplexing strategies (round-robin, rate-of-change).

Our experiments ran on an Intel i7 Nehalem processor with four hardware performance counters. Table 3 lists the machine configuration for this experiment. Note that CPU frequency scaling, which facilitates the CPU in power consumption management, was disabled to make

Platform	*Package*	*Application Domain*	*Description*
PARSEC	blackscholes	Financial Analysis	Calculates portfolio price using Black-Scholes PDE.
	bodytrack	Computer Vision	Tracks a 3D pose of a markerless human body.
	canneal	Engineering	Minimizes routing costs for synthetic chip design.
	vips	Media Processing	Applies a series of transforms to images.
SPLASH.2x	cholesky	HPC	Factors sparse matrix into product of lower triangular matrix.
	fmm	HPC	Simulates interaction in 2D using Fast Multipole method
	ocean_cp	HPC	Large scale movements based on eddy and boundary currents.
	water_spatial	HPC	Evaluates forces and potentials in a system of molecules.

Table 2: PARSEC and SPLASH.2x Benchmarks.

Type	Events
Architecture	Intel Core i7, M640, 2.80 GHz (Nehalem)
Performance counters	IA32_PMC0, IA32_PMC1, IA32_PMC2, IA32_PMC3
Operating system	Linux kernel 3.11.3-rc3
CPU frequency scaling	Disabled
CPU speed	2.8 GHz

Table 3: Machine configuration.

all CPUs run consistently at the same speed (four CPUs running at 2.8 GHz, in our case). The rate-of-change algorithm hooked into `rotate_ctx` (Sec. 2.2.2) and the final counts were exported as a `.csv` file from Perf Event subsystem.

4.1.1 MSE on final counts

We compared our multiplexing technique with the existing round-robin approach by applying a statistical estimator on the mean μ calculated from the single event runs. We used mean-squared error (MSE) for comparing each mean μ_{roc} and μ_{rr} from the multiplexing technique versus the mean μ_{base} from the non-multiplexed run. MSE is defined as follows:

$$MSE = E[(\hat{\theta} - \theta)^2] = B[\hat{\theta}]^2 + var[\hat{\theta}] \qquad (9)$$

4.2 Results

Results show significant improvements for all benchmarks and all events for our rate-of-change approach, when compared with round-robin. Figure 4 compares accuracy in improvement for the two multiplexing strategies. For *instructions*, performance improved with blackscholes (22.1%) when compared with round-robin

Benchmark	Improvement
blackscholes	7.46
bodytrack	5.81
canneal	3.02
vips	1.48
cholesky	4.45
fmm	1.02
ocean_cp	4.67
water_spatial	2.69

Table 4: Improvement (%) per benchmark, averaged over event types.

(8.5%). *cache-misses* had the biggest gain, with close to 95.5% accuracy for both the blackscholes and water_spatial benchmarks. *ref-cpu-cycles* also performed well with rate-of-change (Fig. 4b). Figure 5 shows accuracy rates for *L1-data-cache* events, including *hits*, *misses*, and *prefetches*. For all three events, our rate-of-change approach outperformed round-robin.

Table 4 shows percentage improvements for each benchmark when averaged across event types. The positive improvement rates indicate that our rate-of-change methodology has facilitated in profiling these applications better than the round-robin scheme.

Table 5 shows the improvements for each event type when averaged across benchmarks. Some of the top performers include *instructions*, *cache-misses*, and *node-prefetches*. However, there were also some poor performers. For instance, *branch-misses* had -10.23, while *iL1-misses* had -8.99. System perturbation across different runs may have skewed these numbers. It is possible that round-robin may have been sufficient for scheduling those events, and that our algorithm may have had an effect on those results. These averages only provide an overview of how certain events might behave on a particular benchmark.

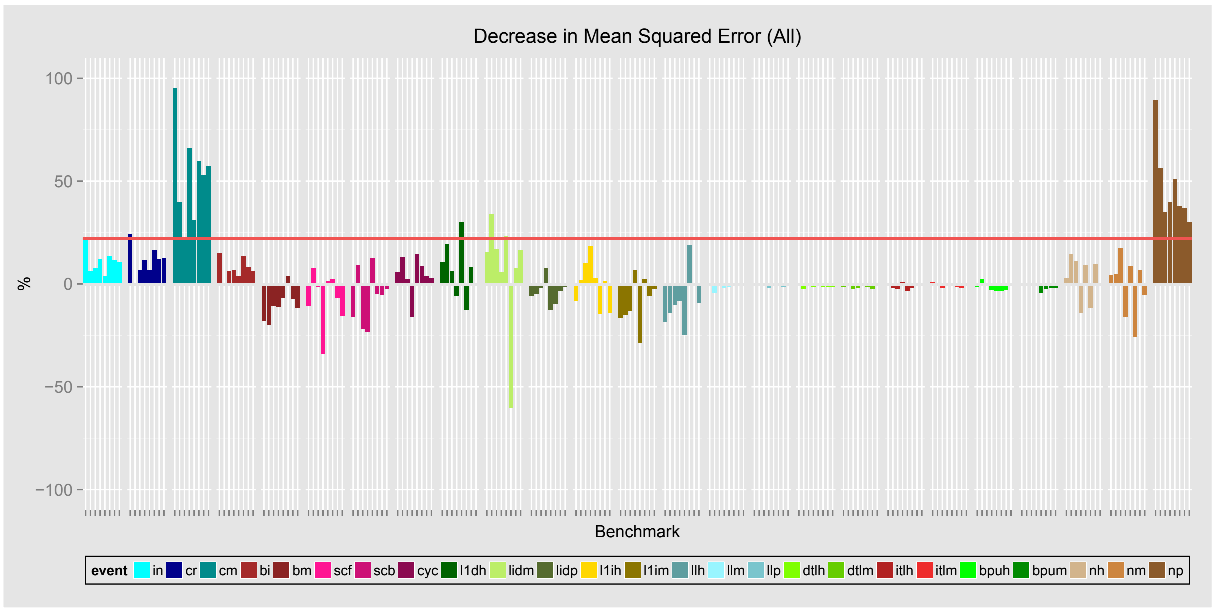


Figure 6: Decrease in MSE for all 25 multiplexed events, when comparing rate-of-change over the round-robin scheduling scheme. Each event set displays improvements for each of the eight benchmarks.

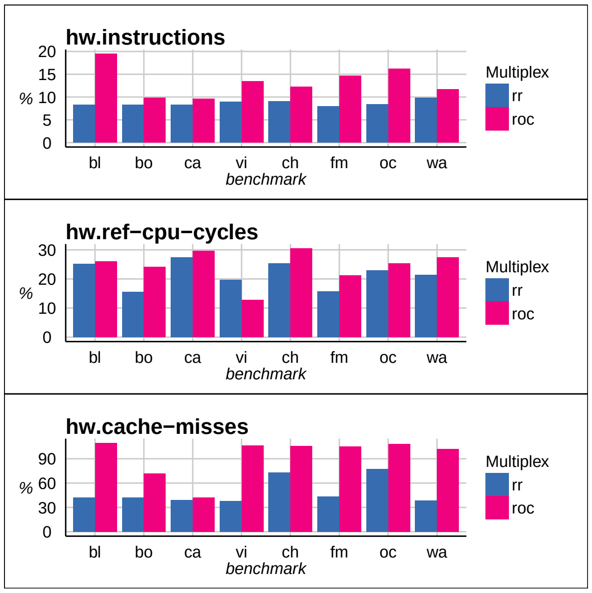


Figure 4: Accuracy in multiplexing strategies (round-robin, rate-of-change) with respect to baseline trace for select hardware events (more is better).

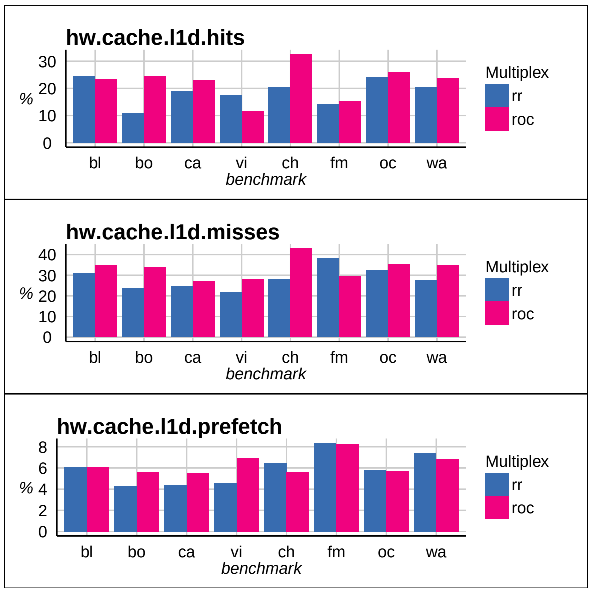


Figure 5: Accuracy in multiplexing strategies (round-robin, rate-of-change) with respect to baseline trace for *L1-data-cache* events (more is better).

instr	*c.ref*	*c.miss*	*br.in*	*br.mi*
10.96	11.35	53.04	7.48	-10.23
iTLB.h	*iTLB.m*	*st.cyc.f*	*st.cyc.b*	*ref.cyc*
-1.11	-0.82	-7.19	-6.44	4.4
dTLB.h	*dTLB.m*	*LLC.h*	*LLC.m*	*LLC.p*
-1.46	-1.59	-8.48	-0.95	-0.95
iL1.h	*iL1.m*	*dL1.h*	*dL1.m*	*dL1.p*
-0.26	-8.99	7.09	7.46	-4.01
bpu.h	*bpu.m*	*node.h*	*node.m*	*node.p*
-1.6	-1.58	2.71	-0.62	47.04

Table 5: Improvement (%) per event type, averaged over benchmark.

Figure 6 shows the proposed methodology's improvement as a decrease in mean-squared error for round-robin versus rate-of-change with respect to the baseline trace. Each event set shows performance for each of the eight benchmarks. The red horizontal line indicates that on average, performance gains of 22% were witnessed when comparing our rate-of-change approach with round-robin. With the exception of vips (*cycles*), cholesky (*iL1-misses*), and fmm (*dL1-misses*), most of the events profiled have shown substantial improvements. Some notable standouts include *cache-misses*: blackscholes (96%), vips (65%), cholesky (32%), fmm (59%); *cache-references*: blackscholes (24%), vips (12%), cholesky (7%), fmm (17%); and *node-prefetches*: blackscholes (89%), vips (40%), cholesky (51%), fmm (37%).

5 Future Work and Conclusion

5.1 Future Work

Our scheduling approach has demonstrated that performance measurement accuracy can increase when incorporating information about the behavior of a program task. Since architectural events are highly correlated in the same run (e.g. hardware interrupts, system calls), one extension to our multiplexing technique would be to incorporate correlated information into calculating the scheduling cost. Information about execution phases, in addition to temporal locality, can facilitate in creating an even more robust scheduler, which can lead to improved profiled accuracy rates. In addition, our multiplexing methodology can serve as an alternative to round-robin scheduling in other areas that utilize real-time decision-making, including task scheduling and decision-support systems.

5.2 Conclusion

Multiplexing multiple performance events is necessary because event counts of the same runs are more correlated to each other than among different runs. In addition, the limited number of m hardware counters provided by architecture manufacturers makes scheduling n performance events impractical, where $n > m$. Our rate-of-change scheduling algorithm has provided an increase of accuracy over the existing round-robin method, improving the precision of the total counts while allowing multiple events to be monitored.

6 References

References

[1] C. Bienia, "Benchmarking Modern Microprocessors," Princeton University, Jan. 2011.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," International Journal of High Performance Computing Applications, 14(3):189-204, June 2000.

[3] L. Campos and I. Scherson, "Rate of Change Load Balancing in Distributed and Parallel Systems," IEEE Symposium on Parallel and Distributed Processing. Apr 1999, doi:10.1109/IPPS.1999.760552.

[4] S. Eranian, "Perfmon2: A Flexible Performance Monitoring Interface For Linux," Proceedings of the Linux Symposium, vol.1, July 2006.

[5] T. Gleixner and D. Niehaus, "Hrtimers and Beyond: Transforming the Linux time Subsystems," Proceedings of the Linux Symposium, vol.1, July 2006.

[6] Intel 64 and IA-32 Architectures Software Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software\-developer-manual-325462.pdf. Intel Corporation, Sept, 2013.

[7] `hrtimer` for Event Multiplexing. https://lkml.org/lkml/2012/9/7/365. Linux Kernel Mailing List. Sept, 2012.

[8] W. Mathur and J. Cook, "Improved Estimation for Software Multiplexing of Performance Counters," 13th IEEE Intl. Symp. on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'05), 2005.

[9] J. May, "MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs," IEEE International Parallel and Distributed Processing Symposium, 2001.

[10] T. Mytkowicz, "Time Interpolation: So Many Metrics, So Few Registers," 40th IEEE/ACM International Symposium on Microarchitecture, 2007, doi:10.1109/MICRO.2007.27.

[11] NVIDIA `nvprof` Profiler. http://docs.nvidia.com/cuda/profiler-users-guide/#axzz33xUbGBm0. NVIDIA CUDA Toolkit v6.0.

[12] AMD Athlon Events. http://oprofile.sourceforge.net/docs/amd-athlon-events.php. OProfile website.

[13] Linux Kernel Profiling with `perf`. https://perf.wiki.kernel.org/index.php/Tutorial. Creative Commons 3.0, 2010.

[14] Qualcomm Adreno Profiler. https://developer.qualcomm.com/mobile-development/maximize-hardware/mobile-gaming-graphics-adreno/tools-and-resources. Qualcomm Developer Network.

[15] S. Shende and A. D. Malony, "The TAU Parallel Performance System," International Journal of High Performance Computing Applications, 20(2):287-311, Summer 2006.

[16] V. Weaver, D. Terpstra and S. Moore, "Non-Determinism and Overcount on Modern Hardware Performance Counter Implementations," ISPASS Workshop, April 2013.

[17] V. Weaver, "Linux perf_event Features and Overhead," 2013.

[18] S. Woo, M. Ohara, E. Torrie, J. Singh, A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," International Symposium on Computer Architecture, 1995, doi:10.1145/223982.223990

The *maxwell(8)* random number generator

Sandy Harris
sandyinchina@gmail.com
sandy.harris@sjtu.edu.cn

Abstract

I propose a daemon process for use on Linux. It gathers entropy from timer calls, distills into a concentrated form, and sends it to the kernel *random(4)* device. The program is small and does not require large resources. The entropy output is of high quality. The output rate varies with the parameters chosen; with the defaults it is about six kilobits per second, which is enough for many applications.

1 Overview

Random numbers are essential for most cryptographic applications, and several otherwise quite good cryptographic systems have been broken because they used inadequate random number generators. The standard reference is RFC 4086, Randomness Requirements for Security [1]. It includes the following text:

> At the heart of all cryptographic systems is the generation of secret, unguessable (i.e., random) numbers.
>
> The lack of generally available facilities for generating such random numbers (that is, the lack of general availability of truly unpredictable sources) forms an open wound in the design of cryptographic software. [1]

However, generating good random numbers is often problematic. The same RFC also says:

> Choosing random quantities to foil a resourceful and motivated adversary is surprisingly difficult. This document points out many pitfalls ... [1]

I will not belabour these points here. I simply take it as given both that high-quality random numbers are important and that generating them can be rather a tricky proposition.

1.1 The Linux random device

Linux provides a random number generator in the kernel; it works by gathering entropy from kernel events, storing it in a pool, and hashing the pool to produce output. It acts as a device driver supporting two devices:

- */dev/random* provides high-grade randomness for critical applications and will block (make the user wait) if the pool lacks entropy
- */dev/urandom* never blocks (always gives output) but is only cryptographically strong, and does not give guaranteed entropy

The main documentation is the manual page, *random(4)*; the source code also has extensive comments. Archives of the Linux kernel mailing list and other lists have much discussion. A critique [16] of an earlier version has been published.

In many situations, the kernel generator works just fine with no additional inputs. For example, a typical desktop system does not do a great deal of crypto, so the demands on the generator are not heavy. On the other hand, there are plenty of inputs—at least keyboard and mouse activity plus disk interrupts.

On other systems, however, the kernel generator may be starved for entropy. Consider a Kerberos server which hands out many tickets, or a system with many encrypted connections, whether IPsec, SSH/TLS or SSH. It will need considerable randomness, but such servers often run headless—no keyboard or mouse—and entropy from disk events may be low. There may be a good deal of network activity, but some of that may be monitored by an enemy, so it is not a completely trustworthy entropy source.

If the kernel generator runs low on entropy, then a program attempting to read */dev/random* will block; the device driver will not respond until it has enough entropy so the user program must be made to wait. A program reading */dev/urandom* will not block but it cannot be certain it is getting all the entropy it expects. The driver is cryptographically strong and the state is enormous, so there is good reason to think the outputs will be of high quality; however, there is no longer a guarantee.

Whichever device they read, programs and users relying on the kernel generator may encounter difficulties if the entropy runs low. Ideally, that would never happen.

1.2 Problem statement

The kernel generator provides an interface that allows an external program to provide it with additional entropy, to prevent any potential entropy shortage. The problem we want to solve here is to provide an appropriate program. The entropy volume need not be large, but the quality should be high.

An essential requirement is that the program not overestimate the entropy it is feeding in, because sufficiently large mis-estimates repeated often enough could cause the kernel generator to misbehave. This would not be easy to do; that generator has a huge state and is quite resilient against small errors of this type. However, frequent and substantial errors could compromise it.

Underestimating entropy is much less dangerous than overestimating it. A low estimate will waste resources, reducing program efficiency. However, it cannot compromise security.

I have written a daemon program which I believe solves this problem. I wanted a name distinct from the existing "Timer entropy daemon" [2], developed by Folkert van Heusden, so I named mine *maxwell(8)*, after Maxwell's demon, an imaginary creature discussed by the great physicist James Clerk Maxwell. Unlike its namesake, however, my program does not create exceptions to the laws of thermodynamics.

2 Existing generators

There are several good ways to get randomness to feed into the kernel generator already. In many—probably even most—cases, one of these will be the best choice and my program will not be necessary. Each of them, however, has disadvantages as well, so I believe there is still a niche which a new program can fill.

Ideally, the system comes with a built-in hardware RNG and failing that, there are other good alternatives. I limit my discussion to three—Turbid, HAVEGE and Cryptlib—each of which has both Open Source code and a detailed design discussion document available. As I see it, those are minimum requirements for a system to inspire confidence.

Also, the authors of all those generators are affiliated with respectable research institutions and have PhDs and publications; this may not be an essential prerequisite for trusting their work, but it is definitely reassuring.

2.1 Built-in hardware

Where it is available, an excellent solution is to use a hardware RNG built into your system. Intel have one in some of their chipsets, Via build one into some CPU models, and so on. If one is buying a server that will be used for crypto, insisting on a hardware RNG as part of your specification is completely reasonable.

The main difficulty of with this method is that not all systems are equipped with these devices. You may not get to choose or specify the system you work on, so the one you have may lack a hardware RNG even if your applications really need one.

Even if the device is present, there will not necessarily be a Linux driver available. In some cases, there might be deficiencies in the documentation required to write a driver, or in the design disclosure and analysis required before the device can be fully trusted.

In short, this is usually the best choice when available, but it is not universally available.

A true paranoid might worry about an intelligence agency secretly subverting such a device during the design process, but this is not a very realistic worry. For one thing, intelligence agencies no doubt have easier and more profitable targets to go after.

Also, if the hardware RNG feeds into *random(4)* then—as long as there is some other entropy—the large driver state plus the complex mixing would make it extremely difficult to compromise that driver even with many of its inputs known. Adding a second good source of entropy—*maxwell(8)*, Turbid or HAVEGE—makes an attack via RNG subversion utterly implausible.

2.2 Turbid

John Denker's Turbid—a daemon for extracting entropy from a sound card or equivalent device, with no microphone attached—is another excellent choice. It can give thousands of output bytes per second, enough for almost any requirement.

Turbid is quite widely applicable; many motherboards include a sound device and on a server, this is often unused. Failing that, it may be possible to add a device either internally if the machine has a free slot or externally via a USB port. Turbid can also be used on a system which uses its sound card for sound. Add a second sound device; there are command-line options which will tell Turbid to use that, leaving the other card free for music, VoIP or whatever.

The unique advantage of Turbid is that it provably delivers almost perfectly random numbers. Most other generators—including mine, *random(4)*, and the others discussed in this section—estimate the randomness of their inputs. Sensible ones attempt to measure the entropy, and are very careful that their estimates are sufficiently conservative. They then demonstrate that, provided that the estimate is good, the output will be adequately random. This is a reasonable approach, but hardly optimal.

Turbid does something quite different. It measures properties of the sound device and uses arguments from physics to derive a lower bound on the Johnson-Nyquist noise [3] which must exist in the circuit. From that, and some mild assumptions about properties of the hash used, it gets a provable lower bound on the output entropy. Parameters are chosen to make that bound 159.something bits per 160-bit SHA context. The documentation talks of "smashing it up against the asymptote".

However, Turbid also has disadvantages. It requires a sound card or equivalent, a condition that is easily satisfied on most systems but may be impossible on some. Also, if the sound device is not already known to Turbid, then a measurement step is required before program parameters can be correctly set. These are analog measurements, something some users may find inconvenient.

The Turbid web page [4] has links to the code and a detailed analysis.

2.3 HAVEGE

The HAVEGE (HArdware Volatile Entropy Gathering and Expansion) RNG gathers entropy from the internal state of a modern superscalar processor. There is a daemon for Linux, *haveged(8)*, which feeds into *random(4)*.

The great advantages of HAVEGE are that the output rate can be very high, up to hundreds of megabits second, and that it requires no extra hardware—just the CPU itself. For applications which need such a rate, it may be the only solution unless the system has a very fast built-in hardware RNG.

However, HAVEGE is not purely a randomness gatherer:

> HAVEGE combines entropy/uncertainty gathering from the architecturally invisible states of a modern superscalar microprocessor with a pseudo-random number generation [5]

The "and Expansion" part of its name refers to a pseudo-random generator. Arguably, this makes HAVEGE less than ideal as source of entropy for pumping into *random(4)* because *any* pseudo-random generator falls short of true randomness, by definition. In this view one should either discard the "and Expansion" parts of HAVEGE and use only the entropy gathering parts, or use the whole thing but give less than 100% entropy credit.

There is a plausible argument on the other side. Papers such as Yarrow [7] argue that a well-designed and well-seeded PRNG can give output good enough for cryptographic purposes. If the PRNG output is effectively indistinguishable from random, then it is safe to treat it as random. The HAVEGE generator's state includes internal processor state not knowable by an opponent and moreover it is continuously updated, so it appears to meet this criterion.

The *haveged(8)* daemon therefore gives full entropy credit for HAVEGE output.

Another difficulty is that HAVEGE seems to be *extremely* hardware-specific. It requires a superscalar processor and relies on:

> a large number of hardware mechanisms that aim to improve performance: caches, branch predictors, ... The state of these components is not architectural (i.e., the result of an ordinary application does not depend on it). [6]

This will not work on a processor that is not superscalar, nor on one to which HAVEGE has not yet been carefully ported.

Porting HAVEGE to a new CPU looks difficult; it depends critically on “non-architectural” features. These are exactly the features most likely to be undocumented because programmers generally need only a reference to the architectural features, the ones that can affect “the result of an ordinary application.”

These “non-architectural” aspects of a design are by definition exactly the ones which an engineer is free to change to get more speed or lower power consumption, or to save some transistors. Hence, they are the ones most likely to be different if several manufacturers make chips for the same architecture, for example Intel, AMD and Via all building x86 chips or the many companies making ARM-based chips. They may even change from model to model within a single manufacturer’s line; for example Intel’s low power Atom is different internally from other Intel CPUs.

On the other hand, HAVEGE does run on a number of different CPUs, so perhaps porting it actually simpler than it looks.

HAVEGE, then, appears to be a fine solution on some CPUs, but it may be no solution at all on others.

The HAVEGE web page [6] has links to both code and several academic papers on the system. The *haveged(8)* web page [15] has both rationale and code for that implementation.

2.4 Cryptlib

Peter Gutmann’s Cryptlib includes a software RNG which gathers entropy by running Unix commands and hashing their outputs. The commands are things like *ps(1)* which, on a reasonably busy system, give changing output.

The great advantage is that this is a pure software solution. It should run on more-or-less any system, and has been tested on many. It needs no special hardware.

One possible problem is that the Cryptlib RNG is a large complex program, perhaps inappropriate for some systems. On the version I have (3.4.1), the random directory has just over 50,000 lines of code (.c .h and .s) in it, though of course much of that code is machine-specific and the core of the RNG is no doubt far smaller. Also the RNG program invokes many other processes so overall complexity and overheads may be problematic on some systems

Also, the RNG relies on the changing state of a multi-user multi-process system. It is not clear how well it will work on a dedicated system which may have no active users and very few processes.

The Cryptlib website [8] has the code and one of Gutmann’s papers [9] has a detailed rationale.

3 Our niche

Each of the alternatives listed above is a fine choice in many cases. Between them they provide quite a broad range of options. What is left for us?

What we want to produce is a program with none of the limitations listed above. It should not impose any hardware requirements, such as

- requiring an on-board or external hardware RNG
- requiring a sound card or equivalent device like Turbid
- requiring certain CPUs as HAVEGE seems to

Nor should it be a large complex program, or invoke other processes, as the Cryptlib RNG does.

Our goal is the smallest simplest program that gives good entropy. I do at least get close to this; the compiled program is small, resource usage is low, and output quality is high.

3.1 Choice of generator

In the most conservative view, only a generator whose inputs are from some inherently random process such as radioactive decay or Johnson-Nyquist circuit noise should be trusted—either an on-board hardware RNG or Turbid. In this view other generators—*random(4)*, *maxwell(8)*, HAVEGE, Cryptlib, Yarrow, Fortuna, ... — are all in effect using system state as a pseudo-random generator, so they cannot be fully trusted. Taking a broader view, any well-designed generator can be used so all those discussed here are usable in some cases; the problem is to choose among them.

If there is a hardware RNG on your board, or HAVEGE runs on your CPU, or you have a sound device free for Turbid—that is the clearly the generator to use. Any of these can give large amounts of high-grade entropy for little resource cost. If two of them are available, consider using both.

If none of those is easily available, the choice is more difficult. It is possible to use *maxwell(8)* in all cases, but using the Cryptlib RNG or adding a device for Turbid should also be considered. In some situations, using an external hardware RNG is worth considering as well.

3.2 Applications for *maxwell(8)*

There are several situations where *maxwell(8)* can be used:

- where the generators listed above are, for one reason or another, not usable
- when using one of the above generators would be expensive or inconvenient
- a second generator run in parallel with any of the above, for safety if the other fails
- when another generator is not fully trusted ("Have the NSA got to Intel?" asks the paranoid)
- whenever a few kilobits a second is clearly enough

There are three main applications:

Using any generator alone gives a system with a single point of failure. Using two is a sensible safety precaution in most cases, and *maxwell(8)* is cheap enough to be quite suitable as the second, whatever is used as the first.

With the *-f* or *-g* option, *maxwell(8)* runs faster and stops after a fixed amount of output. This is suitable for filling up the entropy pool at boot time, or before some randomness-intensive action such as generating a large PGP key.

maxwell(8) can be used even on a very limited systemi—an embedded controller, a router, a plug computer, a Linux cell phone, ... Some of these may not have a hardware RNG, or a sound device that can be used for Turbid, or a CPU that supports HAVEGE. The Cryptlib RNG is not an attractive choice for a system with limited resources and perhaps a cut-down version of Linux that lacks many of the programs that the RNG program calls. In such cases, *maxwell(8)* may be the only reasonable solution.

More than one copy of *maxwell(8)* can be used. The computer I am writing this on uses *haveged(8)* with *maxwell -z* (slow but sure) as a second entropy source and *maxwell -g* for initialisation. This is overkill on a desktop system—probably any of the three would be enough. However, something like that might be exactly what is needed on a busy server.

4 Design overview

The old joke "Good, fast, cheap — pick any two." applies here, with:

good	==	excellent randomness
fast	==	high volume output
cheap	==	a small simple program

I choose good and cheap. We want excellent randomness from a small simple program; I argue that not only is this is achievable but my program actually achieves it.

Choosing good and cheap implies not fast. Some of the methods mentioned above are extremely fast; we cannot hope to compete, and do not try.

4.1 Randomness requirements

Extremely large amounts of random material are rarely necessary. The RFC has:

> How much unpredictability is needed? Is it possible to quantify the requirement in terms of, say, number of random bits per second?
>
> The answer is that not very much is needed. ... even the highest security system is unlikely

> to require strong keying material of much over 200 bits. If a series of keys is needed, they can be generated from a strong random seed (starting value) using a cryptographically strong sequence ... A few hundred random bits generated at start-up or once a day is enough if such techniques are used. ... [1]

There are particular cases where a large burst is needed; for example, to generate a PGP key, one needs a few K bits of top-grade randomness. However, in general even a system doing considerable crypto will not need more than a few hundred bits per second of new entropy.

For example, if a system supports 300 connections and re-keys each of them every 20 minutes, then it will do 900 re-keys an hour, one every four seconds on average. In general, session keys need only a few hundred bits and can get those from */dev/urandom*. Even if each re-key needed 2048 bits and for some reason it needed the quality of */dev/random*, the kernel would need only 512 bits of input entropy per second to keep up.

This would indicate that *maxwell(8)* needs to produce a few hundred bits per second. In fact, it gives an order of magnitude more, a few K bits per second. Details are in the "Resources and speed" section.

4.2 Timer entropy

The paper "Analysis of inherent randomness of the Linux kernel" [10] includes tests of how much randomness one gets from various simple sequences. The key result for our purposes is that (even with interrupts disabled) just:

> doing *usleep(100)*, giving 100 μs delay
> doing a timer call
> taking the low bit of timer data

gives over 7.5 bits of measured entropy per output byte, nearly one bit per sample.

Both the inherent randomness [10] and the HAVEGE [5] papers also discuss sequences of the type:

> timer call
> some simple arithmetic
> timer call
> take the difference of the two timer values

They show that there is also entropy in these. The time for even a simple set of operations can vary depending on things like cache and TLB misses, interrupts, and so on.

There appears to be enough entropy in these simple sequences—either *usleep()* calls or arithmetic—to drive a reasonable generator. That is the basic idea behind *maxwell(8)*. The sequence used in *maxwell(8)* interleaves *usleep()* calls with arithmetic, so it gets entropy from both timer jitter and differences in time for arithmetic.

On the other hand, considerable caution is required here. The RFC has:

> Computer clocks and similar operating system or hardware values, provide significantly fewer real bits of unpredictability than might appear from their specifications.
>
> Tests have been done on clocks on numerous systems, and it was found that their behavior can vary widely and in unexpected ways. ... [1]

My design is conservative. For each 32-bit output, it uses at least 48 clock samples, so if there is 2/3 of a bit of entropy per sample then the output has 32 bits. Then it tells *random(4)* there are 30 bits of entropy per output delivered. If that is not considered safe enough, command-line options allow the administrator to increase the number of samples per output (*-p*) or to reduce the amount of entropy claimed (*-c*) per output.

maxwell(8) uses a modulo operation rather than masking to extract bits from the timer, so more than one bit per sample is possible. This technique also helps with some of the possible oddities in clocks which the RFC points out:

> One version of an operating system running on one set of hardware may actually provide, say, microsecond resolution in a clock, while a different configuration of the "same" system may always provide the same lower bits and only count in the upper bits at much lower resolution. This means that successive reads of the clock may produce identical values even if enough time has passed that the

value "should" change based on the nominal clock resolution. [1]

Taking only the low bits from such a clock is problematic. However, extracting bits with a modulo operation gives a change in the extracted sample whenever the upper bits change.

4.3 Keeping it small

Many RNGs use a cryptographic hash, typically SHA-1, to mix and compress the bits. This is the standard way to distill a lot of somewhat random input into a smaller amount of extremely random output. Seeking a small program, I dispense with the hash. I mix just the input data into a 32-bit word, and output that word when it has enough entropy. Details of the mixing are in a later section.

I also do not use S-boxes, although those can be a fine way to mix data in some applications and are a staple in block cipher design. Seeking a small program, I do not want to pay the cost of S-box storage.

In developing this program I looked an existing "Timer entropy daemon" [2] developed by Folkert van Heusden. It is only at version 0.1. I did borrow a few lines of code from that program, but the approach I took was quite different, so nearly all the code is as well.

The timer entropy daemon uses floating point math in some of its calculations. It collects data in a substantial buffer, 2500 bytes, goes through a calculation to estimate the entropy, then pushes the whole load of buffered data into *random(4)*. My program does none of those things.

maxwell(8) uses no buffer, no hashing, and no S-boxes, only a dozen or so 32-bit variables in various functions. It mixes the input data into one of those variables until it contains enough concentrated entropy, then transfers 32 bits into the random device. The entropy estimation is all done at design time; there is no need to calculate estimates during program operation.

A facility is provided for a cautious system administrator, or someone whose system shows poor entropy in testing, to override my estimates at will, using command-line options, *-p* (paranioa) to make the program use more samples per output or *-c* (claim) to change the amount of entropy it tells *random(4)* that it is delivering. However, even then no entropy estimation is done during actual entropy collection; the user's changes are put into effect when the program is invoked.

It is possible that my current program's method of doing output—32 bits at a time with a *write()* to deliver the data and an *ioctl()* to update the entropy estimate each time—is inefficient. I have not yet looked at this issue. If it does turn out to be a problem, it would be straightforward to add buffering so that the program can do its output in fewer and larger chunks.

The program is indeed small, under 500 lines in the main program and under 2000 overall. SHA-1 alone is larger than that, over 7000 lines in the implementation Turbid uses; no doubt this could be reduced, but it could not become tiny. Turbid as a whole is over 20,000 lines and the Cryptlib RNG over 50,000.

5 Program details

The source code for this program is available from `ftp://ftp.cs.sjtu.edu.cn:990/sandy/maxwell/`. The archive includes a more detailed version of this paper, covering the command-line interface, the internal design of the program, and testing methodologies for evaluating the quality of the output.

6 Analysis

This section discusses the program design in more detail, dealing in particular with the choice of appropriate parameter values.

6.1 How much entropy?

The inherent randomness paper [10] indicates that almost a full bit of entropy can be expected per timer sample. Taking one bit per sample and packing eight of them into a byte, they get 7.6 bits per output byte. Based on that, we would expect a loop that takes 16 samples to give just over 15 bits of entropy. In fact we might get more because *maxwell(8)* uses a modulo operation instead of just masking out the low bit, so getting more than one bit per sample is possible.

I designed the program on the assumption that, on typical systems, we would get at least 12 bits per 16 samples, the number from the inherent randomness [10] paper minus something for safety. This meant it needed

-p	**Options**	**Loops**	**Entropy needed per 16 samples**	
(paranoia)	**(other than -p)**	**2p + 3**	**for 32 bit out**	**for 30 bits claimed**
0	*No options*	3	11	10
1		5	7	6
2	-x	7	5	5
3	-y	9	4	4
4	-z	11	3	3
⋮		⋮	⋮	⋮
7		17	2	2
⋮		⋮	⋮	⋮
15		33	1	1

Table 1: The *-p* (paranoia) option

-p	**Options**	**Loops**	**Entropy needed per 16 samples**	
(paranoia)	**(other than -p)**	**2p + 3**	**for 32 bit out**	**for 30 bits claimed**
0	-f, -g	3	11	6

Table 2: The *-f* or *-g* options

three loops to be sure of filling a 32-bit word, so three is the default.

I also provide a way for the user to override the default where necessary with the *-p (paranoia)* command-line option. However, there is no way to get fewer than three loops, so the program is always safe if 16 samples give at least 11 bits of entropy. The trade-offs are shown in Table 1. With the *-f* or *-g* options, the claim is reduced, as shown in Table 2.

All these entropy requirements are well below the 15 bits per 16 samples we might expect based on the inherent randomness paper [10]. They are also far below the amounts shown by my test programs, described in the previous section. I therefore believe *maxwell(8)* is, at least on systems similar to mine, entirely safe with the default three loops.

In my opinion, setting *-p* higher than four is unnecessary, even for those who want to be cautious. However, the program accepts any number up to 999.

6.2 Attacks

The Yarrow paper [7] gives a catalog of possible weaknesses in a random number generator. I shall go through each of them here, discussing how *maxwell(8)* avoids them. It is worth noting, however, that *maxwell(8)* does not stand alone here. Its output is fed to *random(4)*, so some possible weaknesses in *maxwell(8)* might have no effect on overall security.

The first problem mentioned in [7] is “Entropy Overestimation and Guessable Starting Points”. They say this is both “the commonest failing in PRNGs in real-world applications” and “probably the hardest problem to solve in PRNG design.”

My detailed discussion of entropy estimation is above. In summary, the outputs of *maxwell(8)* have 32 bits of entropy each if each timer sample gives two thirds of a bit. The Inherent Randomness paper [10] indicates that about one bit per sample can be expected and my tests indicate that more than that is actually obtained. Despite that, we tell *random(4)* that we are giving it only 30 bits of entropy per output, just to be safe.

There are also command-line options which allow a system administrator to overrule my estimates. If *maxwell(8)* is thought dubious with the default parameters, try *maxwell -p 3 -c 20* or some such. That is secure if 144 timer samples give 20 bits of entropy.

There is a ”guessable starting point” for each round of output construction; one of five constants borrowed from SHA is used to initialise the sample-collecting variable. However, since this is immediately followed by operations that mix many samples into that variable, it does not appear dangerous.

The next problem mentioned in [7] is "Mishandling of Keys and Seed Files". We have no seed file and do not use a key as many PRNGs do, creating multiple outputs from a single key. Our only key-like item is the entropy-accumulating variable, that is carefully handled, and it is not used to generate outputs larger than input entropy.

The next is "Implementation Errors". It is impossible to entirely prevent those, but my code is short and simple enough to make auditing it a reasonable proposition. Also, there are test programs for all parts of the program.

The next possible problem mentioned is "Cryptanalytic Attacks on PRNG Generation Mechanisms". We do not use such mechanisms, so they are not subject to attack. *random(4)* does use such mechanisms, but they are designed to resist cryptanalysis.

Of course, our mixing mechanism could be attacked, but it seems robust. The QHT is reversible, so if its output is known the enemy can also get its input. However, that does not help him get the next output. None of the other mixing operations are reversible. Because the QHT makes every bit of output depend on every bit of its input, it appears difficult for an enemy to predict outputs as long as there is some input entropy.

The next attacks discussed are "Side Channel Attacks". These involve measuring things outside the program itself—timing, power consumption, electromagnetic radiation, ...—and using those as a window into the internal state.

It would be quite difficult for an attacker to measure *maxwell*'s power consumption independently of the general power usage of the computer it runs on, though perhaps not impossible since *maxwell*'s activity comes in bursts every 100 μs or so. Timing would also be hard to measure, since *maxwell(8)* accepts no external inputs and its only output is to the kernel.

A Tempest type of attack, measuring the electromagnetic radiation from the computer, may be a threat. In most cases, a Tempest attacker would have better things to go after than *maxwell(8)*—perhaps keyboard input, or text on screen or in memory. If he wants to attack the crypto, then there are much better targets than the RNG—plaintext or keys, and especially the private keys in a public key system. If he does go after the RNG, then the state of *random(4)* is more valuable than that of *maxwell(8)*.

However, it is conceivable that, on some systems, data for other attacks would not be available but clock interactions would be visible to an attacker because of the hardware involved. In that case, an attack on *maxwell(8)* might be the best possibility for the attacker. If an attacker using Tempest techniques could distinguish clock reads with nanosecond accuracy, that would compromise *maxwell(8)*. This might in principle compromise *random(4)* if other entropy sources were inadequate, though the attacker would have considerable work to do to break that driver, even with some known inputs.

The next are "Chosen-Input Attacks on the PRNG". Since *maxwell(8)* uses no inputs other than the timer and uses the monotonic timer provided by the Linux real-time libraries, which not even the system administrator can reset, direct attacks on the inputs are not possible.

It is possible for an attacker to indirectly affect timer behaviour, for example by accessing the timer or running programs that increase system load. There is, however, no mechanism that appears to give an attacker the sort of precise control that would be required to compromise *maxwell(8)*—this would require reducing the entropy per sample well below one bit.

The Yarrow paper [7] then goes on to discuss attacks which become possible "once the key is compromised." Since *maxwell(8)* does not use a key in that sense, it is immune to all of these.

Some generators allow "Permanent Compromise Attacks". These generators are all-or-nothing; if the key is compromised, all is lost. Others allow "Iterative Guessing Attacks" where, knowing the state at one time, the attacker is able to find future states with a low-cost search over a limited range of inputs, or "Backtracking Attacks" where he can find previous states. However, *maxwell(8)* starts the generation process afresh for each output; the worst any state compromise could do is give away one 32-bit output.

Finally, [7] mentions "Compromise of High-Value key Generated from Compromised Key". However, even if *maxwell(8)* were seriously compromised, an attacker would still have considerable work to do to compromise *random(4)* and then a key generated from it. It is not clear that this would even be possible if the system has other entropy sources, and it would certainly not be easy in any case.

The program does not use much CPU. It spends most

Option	Delay	Samples	msec per output	K bit/sec
Default	97	48	5	~6
-f, -g	41 or 43	48	2	~16
-x	101	112	11.3	~2.8
-y	103	144	14.8	~2.1
-z	107	176	18.8	~1.6

Table 3: Output rates

of its time sleeping; there is a *usleep(delay)* call before each timer sample, with delays generally around 100 μs. When it does wake up to process a sample, it does only a few simple operations.

The rate of entropy output is adequate for many applications; I have argued above that a few hundred bits per second is enough on most systems. This program is capable of about an order of magnitude more than that. With the default parameters there are 48 *usleep(delay)* calls between outputs, at 97 μs each so total delay is 4.56 ms. Rounding off to 5 ms to allow some time for calculations, we find that the program can output up to 200 32-bit quantities—over six kilobits—per second. Similar calculations for other parameter combinations are shown in Table 3.

Of course, all of these are only approximate estimates. Testing with *dieharder(1)* and a reduced delay shows 367 32-bit rands/sec or 11.7 Kbits/sec, showing that these figures are not wildly out of whack but are likely somewhat optimistic.

On a busy system, the program may be delayed because it is timeshared out, or because the CPU is busy dealing with interrupts. We need not worry about this; the program is fast enough that moderate delays are not a problem. If the system is busy enough to slow this program down significantly for long enough to matter, then there is probably plenty of entropy from disk or net interrupts. If not, the administrator has more urgent things to worry about than this program.

The peak output rate will rarely be achieved, or at least will not be maintained for long. Whenever the *random(4)* driver has enough entropy, it causes any write to block; the writing program is forced to wait. This means *maxwell(8)* behaves much like a good waiter, unobtrusive but efficient. It cranks out data as fast as it can when *random(4)* needs it, but automatically waits politely when it is not needed.

7 Conclusion

This program achieves its main design goal: it uses minimal resources and provides high-grade entropy in sufficient quantity for many applications.

Also, the program is simple enough for easy auditing. The user interface is at least a decent first cut, simple but providing reasonable flexibility.

References

[1] Eastlake, Schiller & Crocker. Randomness Requirements for Security, June 2005. http://tools.ietf.org/html/rfc4086

[2] Timer entropy daemon web page. http://www.vanheusden.com/te/

[3] Wikipedia page on Johnson-Nyquist noise. http://en.wikipedia.org/wiki/Johnson-Nyquist_noise

[4] Turbid web page. http://www.av8n.com/turbid/

[5] A. Seznec, N. Sendrier. HAVEGE: a user level software unpredictable random number generator. http://www.irisa.fr/caps/projects/hipsor/old/files/HAVEGE.pdf

[6] HAVEGE web page. http://www.irisa.fr/caps/projects/hipsor/

[7] J. Kelsey, B. Schneier, and N. Ferguson. Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. *SAC 99*. http://www.schneier.com/yarrow.html

[8] Cryptlib web page. http://www.cs.auckland.ac.nz/~pgut001/cryptlib/

[9] Peter Gutmann. Software Generation of Practically Strong Random Numbers. *USENIX Security Symposium, 1998.* http://www.usenix.org/publications/library/proceedings/sec98/gutmann.html

[10] McGuire, Okech & Schiesser. Analysis of inherent randomness of the Linux kernel. http://lwn.net/images/conf/rtlws11/random-hardware.pdf

[11] James L. Massey. SAFER K-64: A Byte-Oriented Block-Ciphering Algorithm. /em FSE '93.

[12] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, Niels Ferguson. Twofish: A 128-Bit Block Cipher. *First AES Conference.* http://www.schneier.com/paper-twofishpaper.html

[13] Xuejia Lai. On the Design and Security of Block Ciphers. *ETH Series in Information Processing, v. 1, 1992.*

[14] Claude Shannon. Communication Theory of Secrecy Systems. *Bell Systems Technical Journal, 1949.* http://netlab.cs.ucla.edu/wiki/files/shannon1949.pdf

[15] Web page for haveged(8). http://www.issihosts.com/haveged/

[16] Gutterman, Pinkas & Reinman. Analysis of the Linux Random Number Generator. http://eprint.iacr.org/2006/086

Popcorn: a replicated-kernel OS based on Linux

Antonio Barbalace
Virginia Tech
antoniob@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

David Katz
Virginia Tech
dgk8293@vt.edu

Abstract

In recent years, the number of CPUs per platform has continuously increased, affecting almost all segments of the computer market. Because of this trend, many researchers have investigated the problem of how to scale operating systems better on high core-count machines. While many projects have used Linux as a vehicle for this investigation, others have proposed new OS designs. Among them, the replicated-kernel OS model, specifically the multikernel, has gained traction. In this paper, we present Popcorn: a replicated-kernel OS based on Linux. Popcorn boots multiple Linux kernel instances on multicore hardware, one per core or group of cores. Kernels communicate to give to applications the illusion that they are running on top of a single OS. Applications can freely migrate between kernels, exploiting all the hardware resources available on the platform, as in SMP Linux.

1 Introduction

In recent years, the number of CPUs per platform has continuously grown, affecting almost all segments of the computer market. After it was no longer practical to increase the speed of a processor by increasing its clock frequency, chip vendors shifted to exploiting parallelism in order to maintain the rising performance that consumers had come to expect. Nowadays, multiple chips, each containing multiple cores, are being assembled into single systems. All cores, across the different chips, share the same physical memory by means of cache coherency protocols. Although researchers were skeptical that cache coherence would scale [6] the multi core market continues to grow. Multi core processors are ubiquitous, they can be found in embedded devices, like tablets, set top boxes, and mobile devices (e.g., Exynos Octa-Core [3]), in home/office computers (e.g., AMD Fusion, Intel Sandy Bridge), in high-end servers (e.g., AMD Opteron [13], Intel Xeon [1]), and in HPC machines (e.g., SGI Altix [2]). These types of multiprocessor systems, formerly available only as high cost products for the HPC market, are today more affordable and are present in the consumer market. Because of this trend, many researchers have investigated the problem of how to better scale operating systems on high core count machines. While some projects have used Linux as a vehicle for this investigation [6, 7], others have proposed new operating system (OS) designs [5]. Among them, the replicated-kernel OS model has gained traction.

Linux has been extended by its large community of developers to run on multiprocessor shared memory machines. Since kernel version 2.6, preemption patches, ticket spinlocks, read/write locks, and read-copy-update (RCU) have all been added. Several new techniques have also been added to improve data locality, including `per_cpu` infrastructure, the NUMA-aware memory allocator, and support for scheduling domains. B. Wickizer *et al.* [6] conclude that vanilla Linux, on a large-core-count machine, can be made to scale for different applications if the applications are carefully written. In [7] and [12] the authors show that scalable data structures, specifically scalable locks, like MCS, and RCU balanced tree, help Linux scale better when executing select applications.

Although recent research has demonstrated Linux's scalability on multicore systems to some extent, and Linux is already running on high core count machines (e.g., SGI Altix [2]) and accelerators (e.g., Intel Xeon-Phi [15]), it is important to understand whether Linux can be used as the basic block of a replicated-kernel OS. Understanding the advantages of this OS architecture on Linux – not just from a scalability standpoint – is important to better exploit the increasingly parallel hardware that is emerging. If future processors do not provide high-performance cache coherence, Linux's shared memory intensive design may become a significant performance bottleneck [6].

The replicated-kernel OS approach, advanced in operating systems including Hive, Barrelfish, FOS, and others, is a promising way to take advantage of emerging high-core count architectures. A. Baumann *et al.* [5] with Barrelfish introduced the term multikernel OS and showed appealing scalability results demonstrating that their design scales as well, if not better, than SMP Linux on selected applications up to 32 cores. A multikernel OS is an operating system that is made up of different (micro-) kernel instances, each of which runs on a single core of a multi core device. Kernels communicate in order to cooperatively maintain partially replicated OS state. Each kernel runs directly on bare hardware, no (hardware or software) virtualization layer is employed. Because Popcorn does not adhere to such definition we use the term replicated-kernel OS to identify a broader category of multikernels, including Popcorn.

Popcorn In this paper, we present Popcorn: a replicated-kernel OS based on Linux. Popcorn boots multiple Linux kernel instances on multicore hardware, one per core or group of cores, with kernel-private memory and hardware devices. The kernel instances directly communicate, kernel-to-kernel, in order to maintain a common OS state that is partially replicated over every individual kernel instance. Hardware resources (i.e., disks, network interface cards) are fully shared amongst the kernels. Kernel instances coordinate to maintain the abstraction of a single operating system (single system image), enabling traditional applications to run transparently across kernels. Inter-kernel process and thread migration are introduced to allow application threads to transparently execute across the kernels that together form the OS. Considering that vanilla Linux scales well on a bounded number of cores, we do not put any restrictions on how many cores the same kernel image will run on.

Contributions Our primary contribution is an open-source replicated-kernel OS using Linux as its basic building block, as well as its early evaluation on a set of benchmarks. To the best of our knowledge this is the first attempt in applying this design to Linux. Multiple Popcorn kernels, along with the applications that they hosts, can simultaneously populate a multi core machine. To facilitate this, we augmented the Linux kernel with the ability to run within a restricted subset of available hardware resources (e.g. memory). We then strategically partition those resources, ensuring that partitions do not overlap, and dedicate them to single kernel instances.

To create the illusion of a single operating system on top of multiple independent kernel instances we introduced an inter-kernel communication layer, on top of which we developed mechanisms to create a single system image (e.g. single filesystem namespace) and inter-kernel task migration (i.e. task and address space migration and address space consistency). TTY and a virtual network switch was also developed to allow for communication between kernels. Our contribution also includes a set of user-space libraries and tools to support Popcorn. A modified version of *kexec* was introduced to boot the environment; the *util* toolchain was built to create replicated-kernel OS configurations. MPI-Popcorn and the *cthread/pomp* library were introduced to support MPI and OpenMP applications on Popcorn, respectively. Here we describe Popcorn's architecture and implementation details, in particular, the modifications that we introduced into the Linux source code to implement the features required by a replicated-kernel OS design. We also present initial results obtained on our prototype, which was developed on x86 64bit multicore hardware. Popcorn has been evaluated through the use of cpu/memory intensive, as well as I/O intensive workloads. Results are compared to results from the same workloads collected on SMP Linux and KVM.

Document Organization We first present the existing work on the topic in Section 2. We introduce our design choices and architecture in Section 3, and we cover the implementation details of our prototype in Section 4. We present the experimental environment in Section 5 and discuss the results we obtained by running selected benchmarks on Popcorn against mainly vanilla Linux (called *SMP Linux* hereafter) in Section 6. Finally, we conclude in Section 7.

2 Related Work

The body of work related to our approach includes contributions in operating systems, distributed and cluster systems, and Linux design and performance measurements. We leveraged ideas and experience from these different efforts in order to build on their findings and to address their limitations where possible.

Non-Linux based Several decades ago, Hurricane [22] and Hive [11] ('92 and '95) introduced the idea of a replicated-kernel OS by means of clusters or cells (in Hive) of CPUs sharing the same kernel image. This is different from common SMP operating systems, where a single kernel image is shared by all the CPUs. While these approaches were built to work on research hardware, the multikernel model was recently revisited by Barrelfish [5] on modern multicore commodity hardware, with each core loading a single kernel image. A similar approach was taken by FOS [23] addressing emerging high core-count architectures where computational units do not share memory. All these approaches use message passing for inter-kernel communication. These message passing mechanisms are implemented using shared memory programming paradigm. Popcorn follows the same approach but differs in the way the communication mechanism is implemented and in which data structures are kept consistent amongst kernels. Hurricane, Barrelfish and FOS are microkernel-based, but Hive was developed as a modification of the IRIX variant of Unix, similarly to how our work is based on Linux.

Linux-based approaches To the best of our knowledge there is no previous effort that uses Linux as the kernel block in a replicated-kernel OS. However, there are notable efforts in this direction that intersect with the cluster computing domain, including ccCluster (L. McVoy), K. Yaghmour's work [24] on ADEOS, and Kerrighed [18]. Kerrighed is a cluster operating system based on Linux, it introduced the notion of a kernel-level single system image. Popcorn implements a single system image on top of different kernels as well, but the consistency mechanism and the software objects that are kept consistent are fundamentally different. ccCluster and ADEOS aim to run different operating systems on the same hardware on top of a nano-kernel, cluster computing software was used to offer a single system image. Popcorn kernels run on bare hardware.

We know of three separate efforts that have implemented software partitioning of the hardware (i.e. multiple kernel running on the same hardware without virtualization) in Linux before: Twin Linux [16], Linux Mint [19] and SHIMOS [21]. Different from Popcorn, the source code of these solutions are not available (even upon request). Although they describe their (different) approaches, they do not present significant performance numbers, and they do not explore the functionality of their solution over 4 CPUs (on the x86 architecture). Twin Linux was deployed on a dual-core processor. The authors modified GRUB in order to boot two instances of Linux in parallel using different images and hardware resource partitions on different cores. To allow the kernels to communicate with each other, they provide a shared memory area between the kernels. Linux Mint, despite similarities to Twin Linux, lacks an inter-kernel communication facilities. The bootup of different kernel instances in Linux Mint is handled sequentially by the bootstrap processor, something we borrow in our implementation of Popcorn. Popcorn and SHIMOS boot different kernel instances sequentially but any kernel can boot any other kernel. SHIMOS implements an inter-kernel communication mechanism in order to share hardware resources between different Linux instances. Despite the fact that the same functionality is implemented in Popcorn, SHIMOS was designed as a lightweight alternative to virtualization. Therefore it does not implement all of the other features that characterize a multikernel OS.

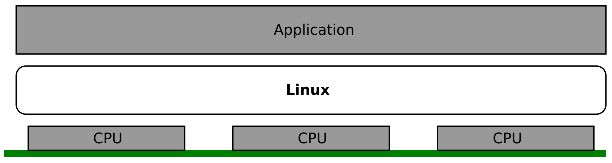


Figure 1: SMP Linux software architecture on multi core hardware. A single kernel instance controls all hardware resources and manages all applications.

3 Popcorn Architecture

The replicated-kernel OS model, mainly suitable for multi-core hardware, implies a different software architecture than the one adopted by SMP Linux. The SMP Linux software stack is depicted in Figure 1. A single kernel instance controls all hardware resources. Applications run in the user space environment that the kernel creates. Popcorn's software stack is shown in Figure 2. Each kernel instance controls a different private subset of the hardware. New software layers have been introduced to allow an application to exploit resources across kernel boundaries. These new software layers are addressed in this section.

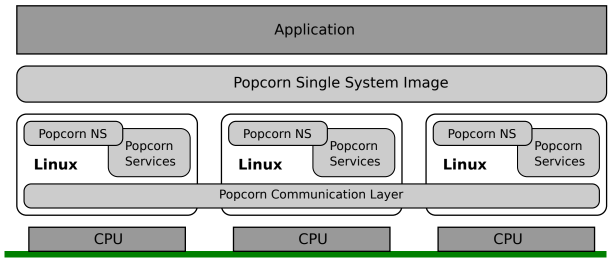


Figure 2: Popcorn Linux software architecture. Each core or group of cores loads a kernel instance. Instances communicate to maintain a single system image.

3.1 Software Partitioning of the Hardware

The idea of running different kernels on the same machine is nowadays associated with virtualization technologies. The replicated-kernel OS model does not imply a virtualization layer. In a virtualized environment different *guest* kernels coexist on top of a hypervisor. Several virtualization solutions (e.g., Xen, KVM) rely on the presence of one of the kernels, the *host*, for services, drawing on a hierarchical relationship between them. There is no hypervisor in our approach. Instead, all kernel instances are peers that reside within different resource partitions of the hardware. Thus, services can run (virtually) anywhere. Without the hypervisor enforcing hardware resource partitioning and managing hardware resource sharing among kernels, the Linux kernel itself should be able to operate with any subset of hardware resources available on the machine. Therefore we added software partitioning of the hardware as first class functionality in Popcorn Linux.

CPUs Within SMP Linux, a single kernel instance runs across all CPUs. After it is started on one CPU, as a part of the initialization process, it sequentially starts all the other CPUs that are present in the machine (Figure 1 and Figure 3.a). Within Popcorn, multiple kernel instances run on a single machine (Figure 2). After an initial kernel, named primary, has been booted up on a subset of CPUs (Figure 3.b), other kernels, the secondaries, can be booted on the remaining CPUs (Figure 3.c and 3.d). Like in SMP Linux each Popcorn kernel instance boots up on single CPU (the bootup) and eventually brings up a group of application CPUs. In Figure 3.b *Processor 0 Core 0* is the bootup CPU and starts *Processor 0 Core 1*. We support arbitrary partitioning and clustering of CPUs, mapping some number of CPUs

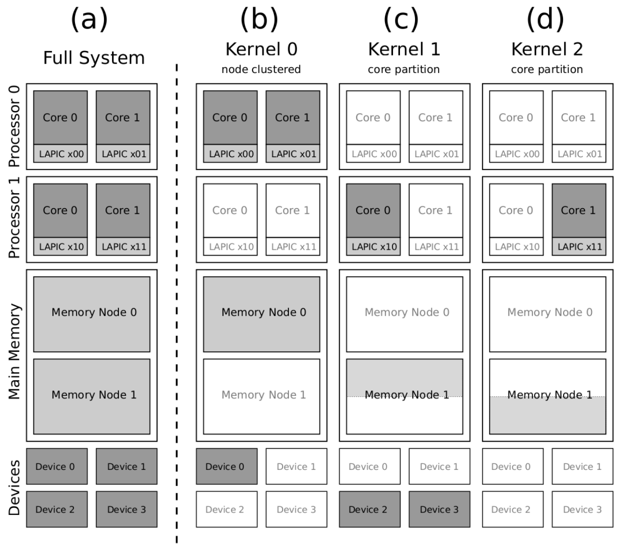


Figure 3: Partitioning and clustering of hardware resources. A kernel instance can be assigned to different a single core or a group of cores. Memory is partitioned on node boundaries (when possible).

to a given kernel instance. Partitioning refers to assigning one kernel instance per processor core (Figure 3.c and 3.d); clustering refers to configurations where a single kernel image runs on multiple processor cores (Figure 3.b). CPUs that are working together in a kernel instance do so in SMP fashion.

Memory and Devices Linux is an SMP OS, as such, when loaded it assumes that all hardware resources must be discovered and loaded; i.e. all resources belong to one kernel instance. In Popcorn, different kernels should coexist, so we start each kernel with a different subset of hardware resources; enforced partitioning is respected by each kernel. Popcorn partitioning includes CPUs (as described above), physical memory and devices. Every kernel owns a private, non overlapping, chunk of memory, and each device is assigned at startup to one only kernel.

In Figure 3 depicts an example of partitioning of 4 devices to 3 kernels. Figure 3.b shows that *Device 0* is owned by *Kernel 0*; Figure 3.c shows that *Device 2* and *Device 3* are own by *Kernel 1*.

On recent multi-core architectures, each group of cores has a certain amount of physical memory that is directly connected to it (or closely bounded). Thus, accessing the same memory area from different processors

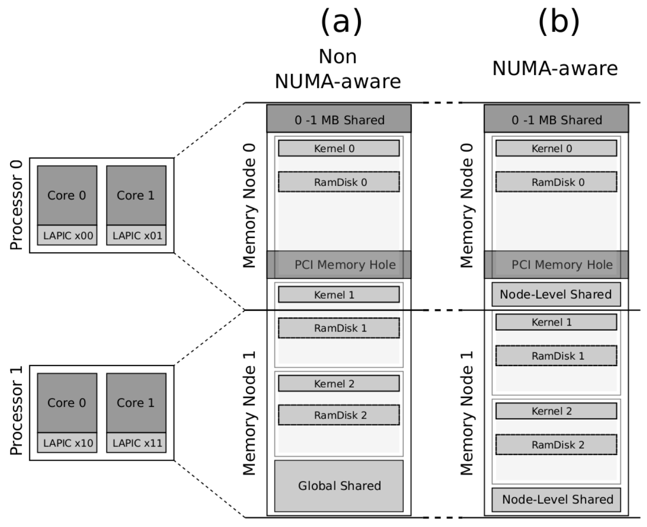


Figure 4: Non-NUMA-aware (a) and NUMA-aware (b) partitioning/clustering of memory on the x86 architecture. The PCI memory hole and the sharing of the first 1 MB of RAM are enforced by the architecture.

incurs different amounts of latencies (NUMA). Physical RAM is usually contiguous but it can contain memory holes, for example to map hardware devices (e.g. PCI hole in x86). Moreover, depending on host architecture, parts of the address space must be shared between all kernels (e.g. in x86 the first 1MB of memory should be shared because of the boot sequence). The partitioning of physical memory must consider all of these architecture-specific details. We developed a variety of memory partitioning policies that apply to CPU partitions or clusters. Figure 4 shows two possible memory policies: non-NUMA-aware (Figure 4.a), and NUMA-aware (Figure 4.b). The latter gives greatest weight to deciding how to allocate private memory windows to the system topology, with the aim of reducing memory access latencies.

3.2 Inter-Kernel Communication

A replicated-kernel OS, strives to provide a single execution environment amongst kernels. In order to accomplish this, kernels must be able to communicate. In Popcorn, communication is based on message passing.

To let the kernels communicate we introduced a low-level message passing layer, deployed over shared memory. Many message passing techniques have been introduced in the literature [9, 10, 5]. We opted for a communication mechanism with private receive-only buffers. Such buffers are allocated in the receiver kernel memory. The layer provides priority-based, synchronous and asynchronous messaging between kernels.

It also provides multicast capabilities to allow for one to many communications. This capability is useful in implementing many distributed algorithms, including distributed locks, voting mechanisms, and commit protocols.

3.3 Single System Image

Applications running on SMP Linux expect a single system image regardless of the CPU they are running on. That is all the CPUs, peripheral devices, memory, can be used by all applications concurrently; furthermore processes communicate and synchronize between them. In Popcorn, the messaging layer is used to coordinate groups of kernels to create a single working environment. Similar to the pioneering work in Plan9 [20], Popcorn's single system image includes: single filesystem namespace (with devices and proc), single process identification (PID), inter-process communication (IPC), and CPU namespaces.

Relative to physical memory and available CPUs, hardware peripherals are comparatively limited in number. After being initialized by a kernel they cannot be accessed in parallel by different kernels; for the same reason concurrent access to devices in SMP Linux is synchronized through the use of spinlocks. Popcorn makes use of a master/worker model for drivers to make devices concurrently accessible through any kernel via the single system image. The master kernel owns the device, and the worker kernels interact with the device through message exchange with the master kernel. A specific example of such a device is the I/O APIC, the programmable interrupt controller on the x86 architecture. The I/O APIC driver is loaded exclusively on the primary kernel, that becomes the master, and any other kernels with a device in their resource partition that requires interrupts registration exchange messages with that kernel in order to operate with the I/O APIC.

3.4 Load Sharing

In a multikernel OS, as in a SMP OS, an applications' threads can run on any of the available CPUs on the hardware. Popcorn was extended to allow user-space tasks to arbitrarily migrate between kernels. Most of the process related data structures are replicated, including the virtual address space. The virtual address space for a process with threads executing on different kernels is maintained consistent over the lifetime of the process.

4 x86 Implementation

Linux was not designed to be a replicated-kernel OS. In order to extend its design, large parts of Linux had to be re-engineered. In this section we cover implementation details that characterize our x86 64bit prototype. We deployed Popcorn starting from vanilla Linux version 3.2.14. The current Popcorn prototype adds a total of $\sim 31k$ lines to the Linux kernel source. The user-space tools are comprised of $\sim 20k$ lines of code, of which $\sim 4k$ lines were added to *kexec*.

4.1 Resource Partitioning

Loading Popcorn Linux requires the establishment of a hardware resource partitioning scheme prior to booting the OS. Furthermore, after the primary kernel has been booted up, all of the remaining kernels can be started. Once this procedure has been followed, the system is ready to be used, and the user can switch to the Popcorn namespace and execute applications on the replicated-kernel OS. Figure 5.a illustrates the steps involved in bringing up the Popcorn system - from OS compilation to application execution. If the *Hotplug* subsystem is available, the steps in Figure 5.b can be followed instead.

In support of partitioning, we have built a chain of applications that gather information on the machine on which Popcorn will run. That information is then used to create the configuration files needed to launch the replicated-kernel environment. This tool chain must be run on SMP Linux, on the target machine, since it exploits resource enumeration provided by the Linux kernel's NUMA subsystem. A vast set of resource partitioning rules can be used to generate the configuration, including and not limited to one-kernel per core, one-kernel per NUMA node, same amount of physical memory per kernel, memory aligned on NUMA boundary. The configuration parameters that are generated are mostly in the form of kernel command-line arguments.

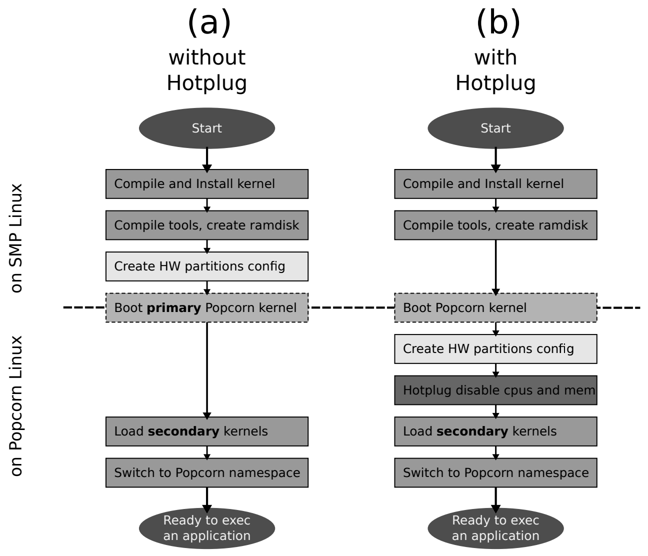

(a)
without Hotplug
(b)
with Hotplug
Start
Compile and Install kernel
Compile tools, create ramdisk
Create HW partitions config
Boot **primary** Popcorn kernel
Boot Popcorn kernel
Create HW partitions config
Hotplug disable cpus and mem
Load **secondary** kernels
Switch to Popcorn namespace
Ready to exec an application
on SMP Linux
on Popcorn Linux

Figure 5: Two different ways of booting Popcorn Linux. Without and with hotplug (CPU and memory) support.

4.2 Booting Multiple Kernels

The *kexec* software is normally used to reboot a machine into a new kernel from an already-running kernel. We modified both the *kexec* application itself and the backend code in the Linux kernel to load new kernel instances (secondary kernels) that run in parallel with the current one, but on a different partition of hardware resources. Kernels are booted in sequence.

As part of the *kexec* project, the Linux kernel was made relocatable, and can potentially be loaded and executed from anywhere within the physical address space [14]. However, it was necessary to rewrite the kernel bootup code in *head_64.S* to boot kernels at any location throughout the entire physical address space. This modification required an addition to the early pagetable creation code on x86 64bit. In order to boot secondary kernels, we modified the trampoline, that is the low level code used to boot application processors in SMP Linux. We added *trampoline_64_bsp.S* whose memory space gets reserved in low memory at boot time. The same memory range is shared by all kernels. Because the trampolines are used by all CPUs, kernels should boot in sequences. The structure `boot_param` has been also modified in order to support a ramdisk sit everywhere in the physical address space.

The boot sequence of a secondary kernel is triggered by a syscall to the *kexec* subsytem. A kernel image is specified in a syscall argument. Rather than reusing the

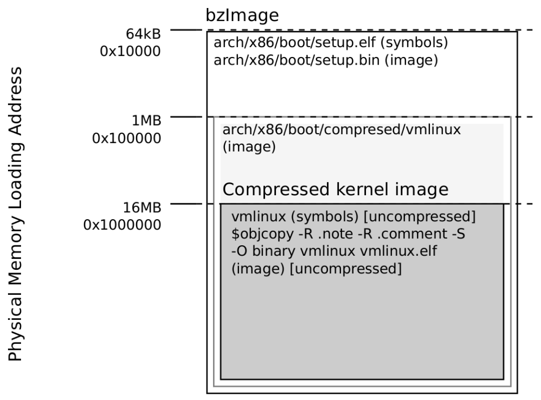

bzImage
Physical Memory Loading Address
64kB
0x10000
arch/x86/boot/setup.elf (symbols)
arch/x86/boot/setup.bin (image)
1MB
0x100000
arch/x86/boot/compresed/vmlinux
(image)
Compressed kernel image
16MB
0x1000000
vmlinux (symbols) [uncompressed]
$objcopy -R .note -R .comment -S
-O binary vmlinux vmlinux.elf
(image) [uncompressed]

Figure 6: Linux's bzImage objects loading addresses. bzImage is a compressed kernel image. It self-extracts the *vmlinux* binary executable at 16MB.

same bzImage used by the primary kernel to bring up the machine, a decompressed and stripped kernel binary is used to boot the secondary kernels. Figure 6 shows the organization of the `bzImage` format. We extract and boot the "Compressed kernel image", i.e. a stripped version of `vmlinux`, to keep the size bounded and speed up the boot process.

The modified *kexec* copies the kernel binary and the boot ramdisk to the selected physical locations. It then initializes the secondary kernel's `boot_params` with the appropriate kernel arguments and ramdisk location/size. At this point *kexec* sets the remote CPU's initial instruction pointer to point to the Popcorn trampoline, and sends an inter-processor interrupt (IPI) to the CPU to wake up. After the remote CPU executes its trampoline, which loads Popcorn' 64bit identity-mapped pagetable for the appropriate region, the CPU is able to start a kernel located anywhere in the physical address space.

4.3 Partitioning CPUs

In SMP Linux, each CPU receives an incremental logical identifier, starting from 0, during boot. This logical id can be acquired by kernel code with a call to `smp_processor_id()`. This identifier is separate from the local APIC identifier adopted in the x86 architecture to distinguish processor cores. In a multicore machine Popcorn tries to keep the same CPU enumeration as is used in SMP Linux; for example CPU identifiers of *Kernel 0* and *Kernel 1* in Figure 3.b and 3.c will be $0,1$ and 2 respectively (not $0,1$ and 0). This was achieved by relying on the APIC ids and the ACPI tables passed in by the BIOS.

We furthermore modified the Linux source to upgrade the legacy subsystems initialization, based on a check on the kernel id, to a more generic mechanism. Many kernel subsystems were initialized only if `smp_processor_id()` was 0. In Popcorn, a new function has been added that returns true if the current CPU is booting up the kernel. Another function, `is_lapic_bsp()`, reveals whether the kernel is the primary.

In order to select which CPUs are assigned to a kernel we introduce the kernel command line argument `present_mask`, which works similarly to `possible_mask` added by the *Hotplug* subsystem. Booting a kernel on a cluster of cores can be done by choosing any combination of core ids. It is not necessary that they are contiguous.

4.4 Partitioning Memory and Devices

A resource-masking feature was implemented to let primary and secondary kernels boot with the same code, on the same hardware. Linux comes with a set of features to include and exclude memory from the memory map provided by the BIOS. We exploited the `memmap` family of kernel command line arguments to accomplish memory partitioning.

In Popcorn, we restrict each kernel to initialize only the devices in its resource partition. When SMP Linux boots, it automatically discovers most of the hardware devices present on the system. This process is possible by means of BIOS enumeration services and dynamic discovery. Dynamic discovery is implemented in several ways, e.g. writing and then reading on memory locations or I/O ports, writing at an address and then waiting for an interrupt. This feature is dangerous if executed by kernel instances other than the kernel that contains a device to be discovered in its resource partition. Currently only the primary kernel has dynamic discovery enabled.

BIOS enumeration services provide lists of most of the devices present in the machine (in x86 ACPI). Each kernel in our system has access to these lists in order to dynamically move hardware resources between running kernels. If the static resource partition of a kernel instance does not include a hardware resource, this resource must not be initialized by the kernel. For example, in the PCI subsystem, we added a blacklisting capability to prevent the PCI driver from initializing a device if it was blacklisted. In our implementation, the blacklist must be provided as a kernel command line argument (`pci_dev_flags`).

4.5 Inter-Kernel Message Passing

A kernel-level message passing layer was implemented with shared memory and inter processor interrupt (UPI) signaling to communicate between kernel instances. The slot-based messaging layer uses cache-aligned private buffers located in the receivers memory. The buffering scheme is multiple writer single reader; concurrency between writers is handled by means of a ticketing mechanism. After a message has been written into a buffer, the sender notifies the receiver with an IPI. In order to mitigate the inter processor traffic due to IPIs, our layer adopts a hybrid of polling and IPI for notification. While the receiver dispatches messages to tasks after receiving a single IPI and message, senders can queue further messages without triggering IPIs. Once all messages have been removed from the receive buffer, IPI delivery is reinstated.

A multicast messaging service is also implemented. In order for a task to send a multicast message it first should open a multicast group. Every message sent to the group is received by the groups subscribers. Multicast groups are opened and closed at runtime. When a message is sent to a group, it is copied to a memory location accessible by all subscribers and an IPI is sent to each of them iteratively. Because in a replicated-kernel OS different kernels coexist on the same hardware, we disable IPI broadcasting (using the kernel command line argument `no_ipi_broadcast`). IPI broadcasting will add additional overhead if used for multicast notification. Nonetheless, the hardware we are using does not support IPI multicast (x2 APIC).

The message passing layer is loaded with `subsys_initcall()`. When loaded on the primary kernel, it creates an array of buffer's physical addresses (`rkvirt`, refer to Figure 7). These arrays are populated by each of the kernels joining the replicated-kernel OS with their respective receiver buffer addresses during their boot processes. The address of this special array is passed via `boot_param` struct to each kernel. Every time a new kernel joins the replicated-kernel OS it adds its receiver buffer address to `rkvirt` first, and then communicates its presence to all the other registered kernels by message.

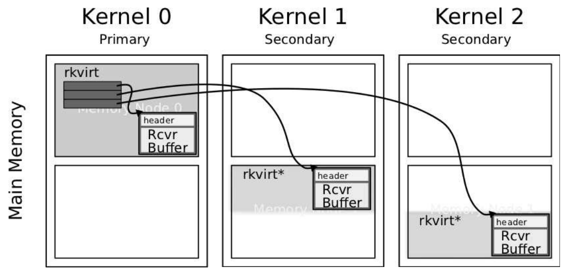


Figure 7: Receiver buffers are allocated in the kernel's private memory. In gray is the private memory of each kernel. `rkvirt`, the array holding the addresses of the receiving buffers, is allocated on the primary kernel.

4.6 Namespaces

Namespaces were introduced into SMP Linux to create sub-environments, as a lightweight virtualization alternative implemented at the OS level. Kerrighed [18] uses namespaces (containers) to migrate applications in a cluster by reproducing the same contained environment on different kernels. Popcorn uses namespaces to provide a single environment shared between kernels. Each kernel's namespace is kept consistent with the others' through the messaging layer.

Linux 3.2.14, on top of which we developed Popcorn, supports uts, mount, IPC, PID and network namespaces, though the API is incomplete. Hence code has been back ported from Linux kernel 3.8. Popcorn was extended to include a *CPU namespace* (see below). Mechanisms were also added to create namespaces that are shared between kernels. Currently the mount namespace relies on NFS. The network namespace is used to create a single IP overlay.

After kernels connect via the messaging layer, static Popcorn namespaces are created in each kernel. A global `nsproxy` structure is then made to point to Popcorn's uts, mount, IPC, PID, network and CPU namespace objects. Namespaces on each kernel are updated whenever a new kernel joins the replicated-kernel OS. Because Popcorn namespaces get created by an asynchronous event rather than the creation of a task, instead of using the common */proc/PID/ns* interface we added */proc/popcorn*. Tasks join Popcorn by associating to its namespaces through the use of the `setns` syscall, which has been updated to work with statically created namespaces. When a task is migrated to another kernel it starts executing only after being (automatically) associated to Popcorn namespaces.

CPU Namespace An application that joins Popcorn's CPU namespace can migrate to any kernel that makes up the replicated-kernel OS, i.e., by mean of `sched_setaffinity()`. Outside Popcorn's CPU namespace, only the CPUs on which the current kernel is loaded are available to applications. If Linux's namespaces provides a subset of the resources (and for CPUs Linux includes cgroups or domains), Popcorn's namespaces show a superset.

We added the kernel command line parameter `cpu_offset`. This is the offset of the current `cpumask_t` in the Popcorn CPU namespace. We added the field `cpuns` of type `struct cpu_namespace *` to the `struct nsproxy`. We augment `task_struct` with a new `list_head` struct to hold a variable length cpu bitmap. Amongst the other, the functions `sched_getaffintiy` and `sched_setaffinity` have been updated to work based on the current CPU namespace. Finally, a task in a different CPU namespace will read different contents from */proc/cpuinfo*. In the Popcorn namespace it will see all the CPUs available in all joining kernels.

4.7 Devices

Following the peer kernel paradigm, implied by the replicated-kernel OS design, we adopted inter-kernel coordination in order to access remotely owned hardware resources. Devices rely on namespaces for enumeration purposes, and message passing for access and coordination. Access to a device can be proxied by a kernel (e.g. the I/O APIC example from Section 3.3) otherwise ownership of a device can be passed to another kernel (in the case where exclusive ownership is required, i.e. CD-ROM burning application). On kernels in which a device is not loaded, a dummy device driver monitors application interaction with that device. Based on the device type, application's requests are either forwarded to the owner kernel (proxied access), or locally staged waiting for local device driver re-initialization (ownership passing).

The following inter-kernel communication devices were implemented outside the messaging dependent mechanism described above for debugging and performance reasons.

Virtual TTY For low-bandwidth applications (e.g. launching processes or debugging) a virtual serial line device, controlled by a TTY driver, is provided between any kernel pair. Each kernel contains as many virtual TTY device nodes (*/dev/vtyX*) as there are kernels in Popcorn (*X* is the smallest CPU id of the kernel to connect to). Each kernel opens a login console on */dev/vty0* during initialization. A shared memory region is divided between all of the device nodes in a bidimensional matrix. Each cell of the matrix holds a ring buffer and the corresponding state variables. The reading mechanism is driven by a timer that periodically moves data from the shared buffer to the flip buffer of the destination kernel.

Virtual Network Switch As in virtual machine environments, we provide virtual networking between kernel instances. The kernel instance that owns the network card acts as the gateway and routes traffic to all the other clients. In this setup each kernel instance has an associated IP address, switching is automatically handled at the driver level. A network overlay, constructed using the network namespace mechanism, provides a single IP amongst all kernels. We developed a kernel level network driver that is based on the Linux TUN/TAP driver but uses IPI notification and fast shared-memory ring buffers for communication. Our implementation uses the inter-kernel messaging layer for coordination and check-in (initialization).

4.8 Task Migration

To migrate tasks, i.e. processes or threads, between kernel instances, a client/server model was adopted. On each kernel, a service is listening on the messaging layer for incoming task migrations. The kernel from which a task would like to out-migrate initiates the communication.

An inter-kernel task migration comprises of three main steps. First, the task which is to be migrated is stopped. Secondly, the whole task state is transferred to the server, where a dummy task, that acts as the migrated task, is created on the remote kernel. Thirdly, all the transferred information about the migrated task are imported into the dummy task, and the migrated task is ready to resume execution.

The task that was stopped on the sending kernel remains behind, inactive, as a *shadow task*. A shadow task is useful for anchoring resources, such as memory and file descriptors, preventing reuse of those resources by the local kernel. Shadow tasks also serve the purpose of speeding up back migrations. When a task is migrated back to a kernel that it has already visited, its current state is installed in the shadow task, and the shadow task is reactivated.

Task State The state of a task is comprised of register contents, its address space, file descriptors, signals, IPCs, users and groups credentials. Popcorn currently supports migrating `struct pt_regs` and `union thread_xstate`, as CPU's registers on the x86 architecture. The address space must also be migrated to support the continued execution of migrated tasks. An address space is comprised of virtual memory area information `struct vm_area_struct`, and the map of physical pages to those virtual memory areas. The latter can obtained by walking the page tables (using `walk_page_range`). Address space mappings are migrated on-demand, in keeping with Linux custom. Mappings are migrated only in response to fault events. This ensures that as a task migrates, overhead associated with carrying out mapping migrations is minimized by migrating only mappings that the task needs. When an application needs a mapping, that mapping is retrieved from remote kernels, and replicated locally. Page level granularity is supported. If no mapping exists remotely, one is created using the normal Linux fault handling routine. This is how tasks come to rely on memory owned by multiple kernels. As a task migrates and executes, it's memory is increasingly composed of locally owned memory pages and remote owned memory pages (the latter do not have an associated `struct page`, therefore are not normal pages). Remote pages are guaranteed to remain available due to the presence of shadow tasks. A configurable amount of address space prefetch was also implemented, and found to have positive performance effect in some situations. Prefetch operations are piggy-backed on mapping retrieval operations to reduce messaging overhead.

State Consistency No OS-level resources are shared between tasks in the same thread group which happen to be running on different kernels. Instead, those resources are replicated and kept consistent through protocols that are tailored to satisfy the requirements of each replicated component.

Inter-kernel thread migration causes partial copies of the same address space to live on multiple kernels. To make multi-threaded applications work correctly on top of these different copies, the copies must never contain conflicting information. Protocols were developed to ensure consistency as memory regions are created, destroyed, and modified, e.g. mmap, mprotect, munmap, etc.

File descriptors, signals, IPCs and credentials are also replicated objects and their state must also be kept consistent through the use of tailored consistency protocols.

5 Evaluation

The purpose of this evaluation is to investigate the behavior of Popcorn when used as 1) a tool for software partitioning of the hardware, and 2) a replicated-kernel OS. A comparison of these two usage modes will highlight the overheads due to the introduced software mechanism for communication, SSI and load sharing between kernels. We compared Popcorn, as a tool for software partitioning of the hardware, to a similarly configured virtualized environment based on KVM, and to SMP Linux. Popcorn as a replicated-kernel OS is compared to SMP Linux.

Hardware We tested Popcorn on a Supermicro H8QG6 equipped with four AMD Opteron 6164HE processors at 1.7GHz, and 64GB of RAM. Each processor socket has 2 physical processors (nodes) on the same die, each physical processor has 6 cores. The L1 and L2 caches are private per core, and 6 MB shared L3 cache exist per processor. All cores are interconnected cross-die and in-die, forming a quasi-fully-connected cube topology [13]. RAM is equally allocated in the machine; each of the 8 nodes has direct access to 8GB.

Software Popcorn Linux is built on the Linux 3.2.14 kernel; SMP Linux results are based on the same vanilla kernel version. The machine ran Ubuntu 10.04 Linux distribution, the ramdisk for the secondary kernels are based on Slackware 13.37.

In a first set of experiments we used the most recent version of KVM/Nahanni available from the project website [17]. We adopted *libvirt* (version 0.10.2) for managing the VMs. Since *libvirt* does not currently support *ivshmem* devices (Nahanni), we modified the *libvirt* source code. To run the MPI experiments, KVM/Nahanni includes a modified version of MPICH2/Nemesis called MPI-Nahanni (although not publicly available). The modified version exploits Nahanni shared memory windows for message passing. The code we received from the authors required some fixes to work properly. Based on MPI-Nahanni we implemented an MPI-Popcorn version of MPICH2/Nemesis.

The OpenMP experiments do not use *glibc/pthread* nor *glibc/gomp*. Instead a reduced POSIX threading library, without futexes, is adopted, called *cthread*. Furthermore *gomp* is replaced by a modified version of the custom OpenMP library derived from the Barrelfish project, we called it *pomp* as Popcorn OpenMP.

In all experiments we setup KVM/Nahanni with one virtual machine per core. Similarly, we configure Popcorn with one kernel per core. Linux runs with all available cores on the machine active but not when running the OpenMP experiments. In this case we set the number of active cores equal to the number of threads.

5.1 CPU/Memory Bound

To evaluate Popcorn on CPU/memory bounded workloads, we used NASA's NAS Parallel Benchmark (NPB) suite [4]. In this paper we present results from Class A versions of Integer Sort (IS), Conjugate Gradient (CG), and Fourier Transform (FT) algorithms. We chose Class A, a small data size, in order to better highlight operating system overheads. NPB is available for OpenMP (OMP) and MPI. The OMP version is designed for shared memory machines, while the MPI version is more suitable for clusters.

Because they use two different programming paradigms, OMP and MPI are not directly comparable. Because of that, we use both versions to quantify the overhead, compared to Linux, of the software partitioning of the hardware functionality, and the full software stack required by the replicated-kernel OS.

MPI A setup in which multiple kernel instances coexist on the same hardware resembles a virtualization environment. Therefore we decided to compare Popcorn, not only with SMP Linux but also with KVM/Nahanni [17] (that resemble the Disco/Cellular Disco replicated-kernel OS solution [8]).

Nahanni allows several KVM virtual machines, each running Linux, to communicate through a shared memory window. MPI applications are used in this test because despite there is shared memory, an OpenMP application can not run across multiple virtual machines. Because this test focuses on compute/memory workloads, we used MPI-Nahanni, which does use network communication, only for coordination, i.e. there is no I/O involved after the application starts. For this test Popcorn is not exploiting it's messaging layer, SSI, or load sharing functionality. MPI-Popcorn relies only on the presence of the standard */dev/mem*, although the virtual network switch is used to start the MPI application.

OpenMP As a replicated-kernel OS, Popcorn is able to transparently run a multithreaded application across multiple kernels. Therefore we compare the performance of the aforementioned NPB applications while running on SMP Linux and Popcorn. OMP NPB applications were compiled with *gcc* once with *cthread*, and *pomp*, and then run on both OSes. This experiment highlights the overhead due to all of Popcorn's software layers.

5.2 I/O Bound

This test investigates the response time of a web server running on our prototype in a secondary kernel. Because our inter-kernel networking runs below the SSI layer, this test stresses the software partitioning of the hardware functionality of Popcorn.

We run the event-based *nginx* web server along with ApacheBench, a page request generator from the Apache project. From a different machine in the same network, we generate http requests to SMP Linux, to secondary kernels in Popcorn, and to guest kernels on KVM. This configuration is shown in Figure 8. Although higher-performance network sharing infrastructures exist for KVM, we use the built-in bridged networking that is used in many research and industrial setups.

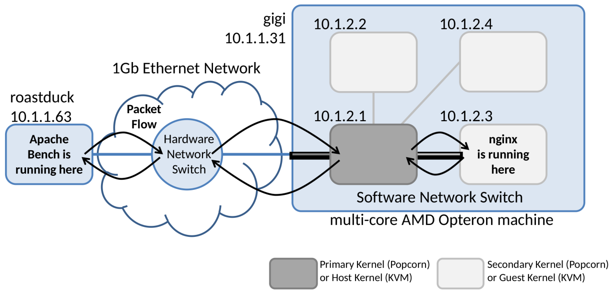


Figure 8: The network configuration used to generate the results in Figure 15.

6 Results

6.1 CPU/Memory workloads

MPI Figures 9, 10 and 11 show how Popcorn and its competitors perform on integer sort, the Fourier transform, and the conjugate gradient benchmarks, respectively, within MPI. Graphs are in log_2 scale. For each data point, we ran 20 iterations, and provide average and standard deviation. Popcorn numbers are always close to SMP Linux numbers, and for low core counts Popcorn performs better. In the best case (on 2 and 4 cores running CG), it is more than 50% faster than SMP Linux. In the worst case (on 16 cores running FT), Popcorn is 30% slower the SMP Linux.

In Linux, MPI runs a process per core, but such processes are not pinned to the cores on which they are created: they can migrate to another less loaded core if the workload becomes unbalanced. In Popcorn, each process is pinned by design on a kernel running on a single core; if the load changes, due to system activities, the test process can not be migrated anywhere else. This is part of the cause of the observed SMP Linux's trends. Popcorn must also pay additional overhead for virtual networking and for running the replicated-kernel OS environment. We believe that communication via the virtual network switch is the main source of overhead in Popcorn. Considering the graphs, this overhead appears to be relatively small, and in general, the numbers are comparable. Finally, Popcorn enforced isolation results in a faster execution at low core counts; this shows that Linux's distributed work stealing scheduling algorithm can be improved.

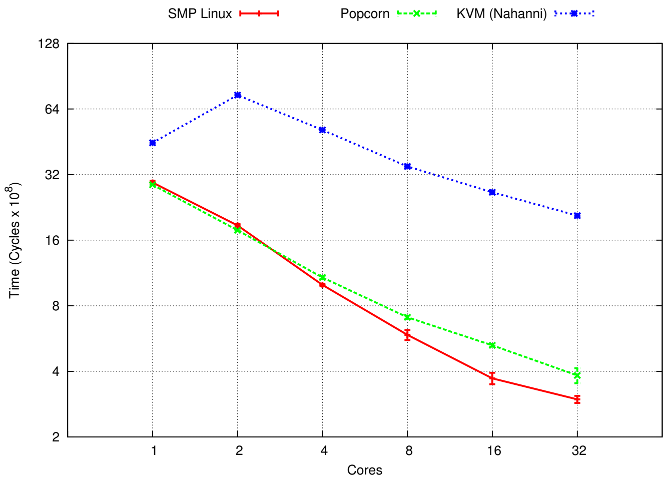


Figure 9: NPB/MPI integer sort (IS) benchmark results.

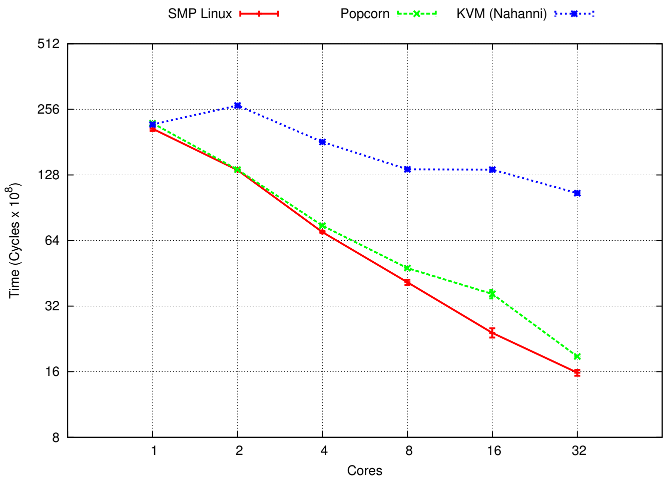


Figure 10: NPB/MPI fast Fourier transform (FT) benchmark results.

Nahanni is the worst in terms of performance. It is up to 6 times slower than SMP Linux and Popcorn (on 32 cores running either CG or FT). Although Nahanni's one core performance is the same as SMP Linux, increasing the core count causes the performance to get worse on all benchmarks. A slower benchmark execution was expected on Nahanni due to the overhead incurred by virtualization. However, the high overhead that was observed is not just because of virtualization but is also due to inter-VM communication and scheduling on the Linux host.

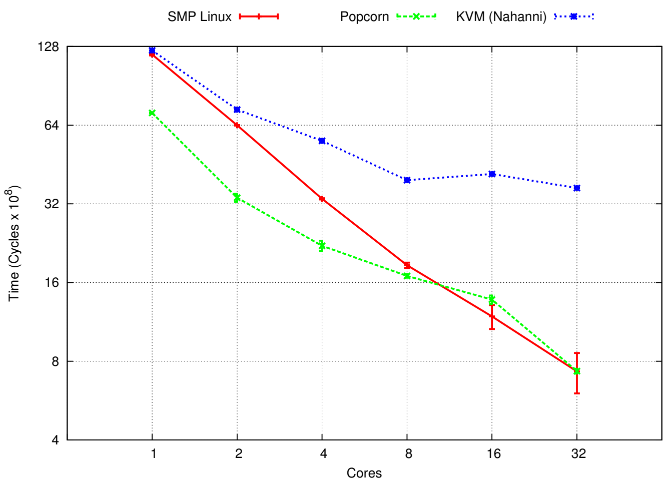


Figure 11: NPB/MPI conjugate gradient (CG) benchmark results.

OpenMP Figures 12, 13 and 14 show how Popcorn and SMP Linux perform on IS, FT, and CG benchmarks, respectively, within OpenMP. Graphs are in log_2 scale. For each data point, we ran 20 iterations, and we provide average and standard deviation.

Unlike the MPI experiments, there is no obvious common trend among the experiments. In the IS experiment, Popcorn is in general faster than SMP Linux, SMP Linux is up to 20% slower (on 32 cores). The FT experiment shows similar trends for Popcorn and SMP Linux, although SMP Linux is usually faster (less than 10%) but not for high core counts. In the CG experiment Popcorn performs poorly being up to 4 times slower than SMP Linux on 16 cores.

These experiments show that the performance of Popcorn depends on the benchmark we run. This was expected, as our address space consistency protocol performance depends on the memory access pattern of the application. Analysis and comparison of the overhead in SMP Linux and Popcorn Linux reveals that the removal of lock contention from SMP Linux yields significant gains for Popcorn over SMP Linux. This contention removal is an artifact of the fact that data structures are replicated, and therefore access to those structures does not require significant synchronization. However, Popcorn must do additional work to maintain a consistent address space. These two factors battle for dominance, and depending on the workload, one will overcome the other. Mechanisms are proposed to reduce the Popcorn overhead with the goal of making further performance gains on SMP Linux.

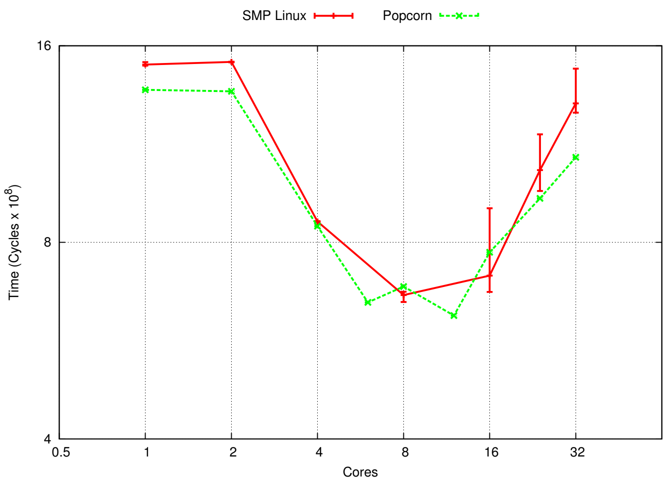


Figure 12: NPB/OpenMP integer sort (IS) benchmark results.

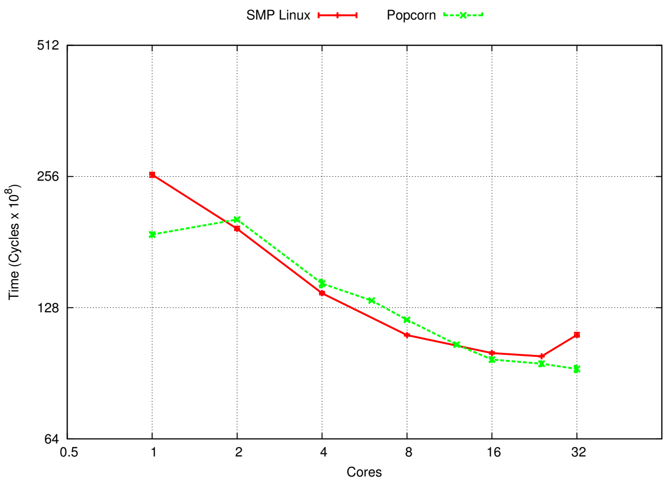


Figure 13: NPB/OpenMP fast Fourier transform (FT) benchmark results.

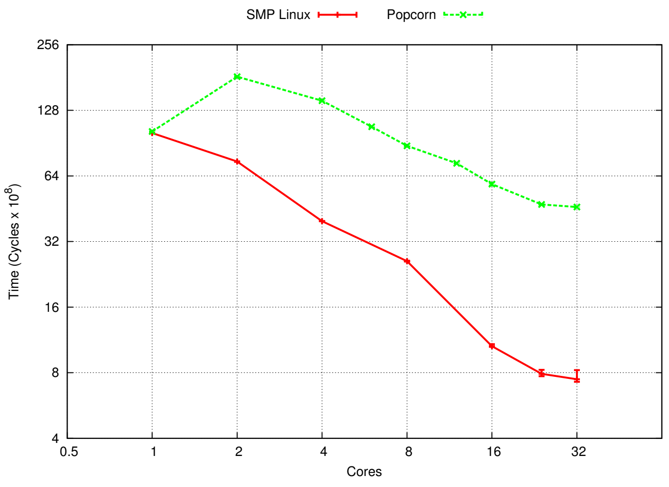


Figure 14: NPB/OpenMP conjugate gradient (CG) benchmark results.

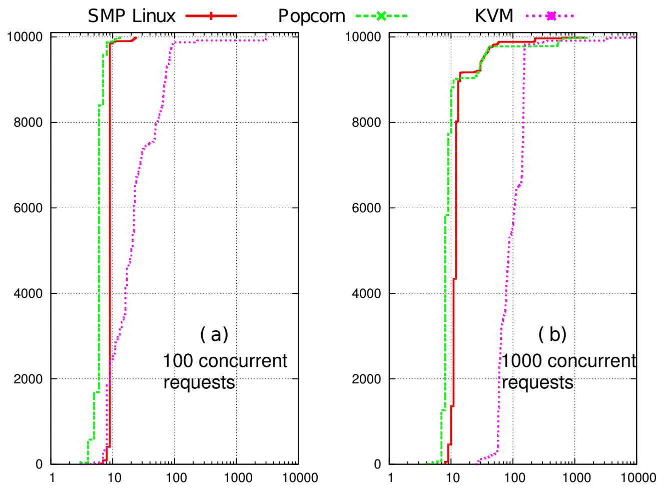


Figure 15: *Apache Bench* results on *nginx* web server on SMP Linux, Popcorn and KVM. Response time in *ms* on the *x* axis, number of requests on the *y* axis.

6.2 I/O Bound

We set ApacheBench to run 10000 requests with a concurrency level of 100 and 1000 threads to obtain the data in Figure 15.

The data show that the Popcorn architecture does not hinder the performance of running a web server (I/O bound benchmark) on the selected hardware, when compared to SMP Linux. Figure 15 shows that in most cases, Popcorn can serve a request faster then Linux, which is attributable both to scheduling and to the fact that the task of handling the hardware network interface is shared with the kernel instance that owns it. KVM suffers due to virtualization and scheduling overhead.

7 Conclusions

We introduced the design of Popcorn Linux and the re-engineering required by the Linux kernel to boot a coexistent set of kernels, and make them cooperate as a single operating system for SMP machines. This effort allowed us to implement and evaluate an alternative Linux implementation while maintaining the shared memory programming model for application development.

Our project contributes a number of patches to the Linux community (booting anywhere in the physical address space, task migration, etc.). Popcorn's booting anywhere feature offers insight into how to re-engineer Linux subsystems to accommodate more complex bootup procedures. Task migration enables Linux to live migrate execution across kernels without virtualization support. Next steps include completing and consolidating the work on namespaces, using rproc/rpmsg instead of *kexec* and re-basing the messaging on *virtio*.

Our MPI results show that Popcorn provides results similar to Linux, and that it can outperform virtualization-based solutions like Nahanni by up to a factor of 10. The network test shows that Popcorn is faster than Linux and Nahanni. Popcorn is not based on hypervisor technologies and can be used as an alternative to a set of virtual machines when CPUs time-partitioning is not necessary. The OpenMP results show that Popcorn can be faster, comparable to, or slower then Linux, and that this behaviour is application dependent. The replicated-kernel OS design applied to Linux promises better scaling, but the gains are sometimes offset by other source of overheads (e.g. messaging). An analysis on a thousands of cores machine can provide more insight into this solution.

It turns out that a replicated-kernel OS based on Linux aggregates the flexibility of a traditional single-image OS with the isolation and consolidation features of a virtual machine, but on bare metal, while being potentially more scalable on high core count machines. The full sources of Popcorn Linux and associated tools can be found at http://www.popcornlinux.org.

Acknowledgments

This work would not have been possible without the help of all the people involved in the Popcorn project. We are particularly thankful to Ben Shelton, Sean Furrow, Marina Sadini, Akshay Ravichandran, Saif Ansary, and Alastair Murray for their contributions. Moreover we are thankful to Cam MacDonell at the University of Alberta for helping us install the Nahanni code and giving us access to their private source code.

References

[1] An introduction to the Intel quickpath interconnect. 2009.

[2] Technical advances in the SGI UV architecture, 2012.

[3] Samsung exynos 4 quad processor, 2013.

[4] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D.

Simon, V. Venkatakrishnan, and S. K. Weeratunga. The nas parallel benchmarks summary and preliminary results. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new OS architecture for scalable multicore systems. SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[6] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. An analysis of Linux scalability to many cores. OSDI'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[7] Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, July 2012.

[8] Edouard Bugnion, Scott Devine, Kinshuk Govil, and Mendel Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. *ACM Trans. Comput. Syst.*, 15(4):412–447, November 1997.

[9] D. Buntinas, G. Mercier, and W. Gropp. Design and evaluation of nemesis, a scalable, low-latency, message-passing communication subsystem. In *Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on*, volume 1, pages 10 pp. –530, may 2006.

[10] Darius Buntinas, Brice Goglin, David Goodell, Guillaume Mercier, and Stéphanie Moreaud. Cache-efficient, intranode, large-message MPI communication with MPICH2-Nemesis. In *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, pages 462–469, Washington, DC, USA, 2009. IEEE Computer Society.

[11] John Chapin and et al. Hive: Fault containment for shared-memory multiprocessors, 1995.

[12] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. Scalable address spaces using rcu balanced trees. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 199–210, New York, NY, USA, 2012. ACM.

[13] P. Conway, N. Kalyanasundharam, G. Donley, K. Lepak, and B. Hughes. Cache hierarchy and memory subsystem of the AMD Opteron processor. *Micro, IEEE*, 30(2):16 –29, march-april 2010.

[14] Vivek Goyal. ELF relocatable x86 bzimage (v2), 2006.

[15] Intel Corporation. Xeon Phi product family. http://www.intel.com/content/www/us/en/processors/xeon/ xeon-phi-detail.html.

[16] A. Kale, P. Mittal, S. Manek, N. Gundecha, and M. Londhe. Distributing subsystems across different kernels running simultaneously in a Multi-Core architecture. In *Computational Science and Engineering (CSE), 2011 IEEE 14th International Conference on*, pages 114–120, 2011.

[17] Xiaodi Ke. Interprocess communication mechanisms with Inter-Virtual machine shared memory. Master's thesis, University of Alberta, August 2011.

[18] C. Morin, P. Gallard, R. Lottiaux, and G. Vallee. Towards an efficient single system image cluster operating system. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 370–377, Oct 2002.

[19] Yoshinari Nomura, Ryota Senzaki, Daiki Nakahara, Hiroshi Ushio, Tetsuya Kataoka, and Hideo Taniguchi. Mint: Booting multiple linux kernels on a multicore processor. pages 555–560. IEEE, October 2011.

[20] Rob Pike, David L. Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. Plan 9 from bell labs. *Computing Systems*, 8(2):221–254, 1995.

[21] T. Shimosawa, H. Matsuba, and Y. Ishikawa. Logical partitioning without architectural supports. In *Computer Software and Applications, 2008. COMPSAC '08. 32nd Annual IEEE International*, pages 355–364, July 2008.

[22] Ron Unrau, Orran Krieger, Benjamin Gamsa, and Michael Stumm. Hierarchical clustering: A structure for scalable multiprocessor operating system design. *Journal of Supercomputing*, 9:105–134, 1993.

[23] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43(2):76–85, 2009.

[24] Karim Yaghmour. A practical approach to linux clusters on smp hardware. Technical report, 2002.