A Dataflow Runtime Environment and Static Scheduler for Edge, Fog and In-Situ Computing

Caio B. G. Carvalho, Victor C. Ferreira, Felipe M. G. França

Programa de Engenharia de Sistemas e Computação - COPPE
Universidade Federal do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
Phone: +55 21 3938-8672 E-mail: {cbgc,vcruz,felipe}@cos.ufrj.br

Cristiana B. Bentes

Departamento de Engenharia de Sistemas e Computação
Faculdade de Engenharia Universidade do Estado do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
E-mail: cris@eng.uerj.br

Gabriele Mencagli

Department of Computer Science
University of Pisa, Pisa, Italy
Phone: +39-050-2213132 E-mail: mencagli@di.unipi.it

Tiago A. O. Alves, Alexandre C. Sena, Leandro A. J. Marzulo

Departamento de Informática e Ciência da Computação
Instituto de Matemática e Estatística Universidade do Estado do Rio de Janeiro, Rio de Janeiro, RJ, Brazil
Phone: +55 21 2334-0144 E-mail: {tiago,asena,leandro}@ime.uerj.br

Abstract: In the dataflow computation model, instructions or tasks are executed according to data dependencies, instead of following the program order, thus allowing natural parallelism exploitation. A wide variety of dataflow-based solutions, in different flavors and abstraction levels (from processors to runtime libraries), have been proposed as interesting alternatives for harnessing the potential of modern computing systems. Sucuri is a dataflow library for Python that allows users to specify their application as a dependency graph and execute it transparently at clusters of multicores, while taking care of scheduling issues. Recent trends in Fog and In-situ computing assume that storage and network devices will be equipped with processing elements that usually have lower power consumption and performance. An important decision on such system is whether to move data to traditional processors (paying the communication costs), or performing computation where data is sitting, using a potentially slower processor. Hence, runtime environments that deal with that trade-off are extremely necessary. This work presents a study on different factors that should be considered when running dataflow applications in Edge/Fog/In-situ environment. We use Sucuri to manage the execution in a small system with a regular PC and a Parallella board, emulating a smart storage (Edge/Fog/In-situ device). Experiments performed with a set of benchmarks show how data transfer size, network latency and packet loss rates affect execution time when outsourcing computation to the smart storage. Then, a static scheduling solution is presented, allowing Sucuri to avoid outsourcing when there would be no performance gains.

Keywords: Dataflow Computing; Edge Computing; Fog Computing; Scheduling Techniques; Smart Storage.

1 Introduction

Parallel programming is paramount for fully harvesting the available computational power of modern architectures, which are often composed by different co-processors and accelerating devices, including GPUs, Xeon Phi processors and FPGAs (Caulfield et al. 2016). Moreover, trending applications related to Deep

Neural Networks and Internet-of-Things produce an ever increasing amount of data that needs to be efficiently stored, usually in a distributed way. When devising such applications, programmers need to consider the impact on performance caused by data movements between storage/memory devices and processing elements.

Edge/Fog/In-situ computing (OpenFog Consortium Architecture Working Group 2016, Shi et al. 2016) proposes bringing computation closer to where the data is sitting, by adding computational capabilities to storage devices (Jun et al. 2015, Kim et al. 2011, NGD Systems 2017), network devices (Juniper 2018) (such as NICs, switches and routers) or even using mobiles devices. Those "smart" devices would be able to perform part of the computation that would reduce data transmission over the network and data buses. Moreover, those devices could be equipped with processors custom-made for the application, which could result in good performance, even for low power systems.

All the aforementioned issues require a proper tool-set that helps programmers devise parallel and distributed applications that can be executed on a large spectrum of devices. Furthermore, such tools should shield developers from dealing with technological aspects pertaining task creation, synchronisation and edge/fog/in-situ aware scheduling.

The Dataflow programming model seems to be a good candidate for edge/fog/in-situ applications, since it provides a simple and natural way of exploiting parallelism. A dataflow program is usually represented as a directed graph, where tasks (or instructions) are depicted by nodes and data dependencies are denoted by edges between nodes. In dataflow, instructions or tasks are allowed to run as soon as their input operands are ready, instead of following program order. Independent tasks can be naturally identified and executed in parallel, if there are enough resources. Dataflow-based APIs and runtime environments (Alves et al. 2011, Marzulo et al. 2014, TBB 2014, Wozniak et al. 2013, Wilde et al. 2011, Matheou & Evripidou 2016) can be used on top of Von Neumann architectures with performance equivalent to well-known tools for parallel programming such as OpenMP or Pthreads. In addition, increasing the granularity level of the dataflow programming can turn it into a good coordination language. It is easier to outsource blocks of computation, like functions, to different cores and machines (Johnston et al. 2004). IoT nodes can also be abstracted as nodes in a graph flow, as stated in (Giang et al. 2015). The library or runtime in use could take advantage on that to distribute the work among the devices.

In this work, we continue the study presented in (Carvalho et al. 2017), aiming at using the dataflow model for devising edge/fog/in-situ applications. Since all data dependencies are explicitly described in the graph, dataflow runtime environments could make use of that information to schedule task execution according to edge/fog/in-situ demands. Our proposal employs Sucuri (Alves et al. 2014, Sena et al. 2015, Silva et al. 2016) to orchestrate edge/fog/in-situ devices that come into play. Sucuri is a dataflow library for Python that provides an easy interface for parallel programming where developers can accommodate custom functions into nodes and only fill in the dependencies connecting them with edges inside a graph. Sucuri also creates an abstraction layer that uses MPI for communicating with remote machines in a transparent way. As Sucuri has already been developed and in use by our research group, and presented good performance with ease of programming, it would be a natural candidate for our approach.

To emulate a smart storage device we used a Parallella board (Parallella 2014), equipped with a Xilinx Zynq Z7010 (ARM Cortex A9 dual core + FPGA) (Xilinx 2017) and a Epiphany 16-core RISC processor. The Parallella board runs a Linux operating system on a SD Card where input files of our benchmarks are also stored. A traditional computer (PC) communicates with the Parallela board through an ethernet connection. In this version, the FPGA and Epiphany were not used.

Our study consists in evaluating the impact of employing our solution in scenarios with different network latencies and packet loss rates, using different applications and input file sizes. With that information in hand, this work proposes an improvement on the static scheduling mechanism for Sucuri that tries to predict transfer and execution times, to decide whether it is better to outsource computation to the (slower) smart storage processor, or move data from disk to the (more powerful) PC. Our benchmarks include two artificial benchmarks (text processing applications) and a set of search and ordering algorithms. Results show that, even for a low power device with limited computational capabilities, it is possible to obtain speedups by avoiding unnecessary data transfers. Gains are maximised when task computational costs are low and input sets are larger. Moreover, scenarios with higher network latency and packet loss make our approach more appealing. Finally, our proposed static scheduling mechanism was able to make good decisions and avoid outsourcing work when this would affect performance.

This paper is organised as follows. Section 2 consists of related work in edge/fog/in-situ data processing. Section 3 presents an overview of the Sucuri dataflow library for Python. Section 4 presents the changes made in Sucuri for edge/fog/in-situ environments. Section 5 presents the experimental analysis. At last, Section 6 discusses what can be taken from the results found in the previous section and also what can be done as future work.

2 Related Work

In this section, we review some research papers closely related to Sucuri. As stated in prior work (Giang et al. 2015), the dataflow model is a valuable candidate to execute Fog/In-Situ computations provided that some

properties are met by new runtime systems: notably, they need support to device heterogeneity, adaptability and scalability. Although Sucuri does not have all these properties yet, it goes on the right direction.

2.1 Dataflow runtime systems

The dataflow model is a widely studied programming model for expressing parallelism driven by pure data dependencies among tasks. Several runtime systems have adopted this model by exposing different interfaces and optimised scheduling strategies.

Corral (Jalaparti et al. 2015) is a framework that takes advantage of predictability of recurring jobs in computer clusters to perform offline scheduling. The framework tries to place computation and data jointly to improve locality. Moreover, in order to reduce interference between jobs, it performs spatial and temporal isolation. The scheduler will try to place jobs in different regions of the cluster and will try to avoid concurrent execution of certain jobs. Corral was implemented on Apache Yarn and executed on a 210 machine cluster, reducing the makespan of production workloads of up to 33% when compared to Yarn's capacity scheduler. Moreover, there was a reduction of the average completion time of up to 56% and 20-90% of reduction in data transferred across racks. Corral is not suitable to target Fog/In-Situ computing scenarios since it targets clusters of homogeneous nodes. However, its scheduling policies are highly optimised and could be implemented in any dataflow runtime including Sucuri.

Swift/T (Wozniak et al. 2013) is a description language and runtime that supports the dynamic creation of workflows with different task granularity and execution on platforms with a huge number of processing elements. Swift/T employs Asynchronous Dynamic Load Balancing (ADLB) to distribute tasks among computing nodes. However, task data sharing is done through a parallel file system that can cause performance degradation, due to poor data locality and interference with other applications. In (Duro et al. 2014), authors propose exploiting data locality in Swift/T applications through Hercules, a distributed in-memory store based on Memcached (Fitzpatrick 2004). Swift/T implementation was optimised to schedule computation jobs in the nodes where required data is stored. The proposal was evaluated using a synthetic application that accesses raw files with different patterns, showing promising results. Although interesting, Swift/T supports in-memory key-value stores to allow data to be shared among tasks, while dataflow runtimes for Fog/In-Situ computing should be efficiently based on a persistent storage to address the high dynamicity and possibly unreliability of IoT computing environments.

The dataflow model has been adopted in the design of several runtime systems for parallel computing. The ffMDF runtime (Buono et al. 2014) provides a lightweight support for dense linear algebra kernels. It has a dynamic macro-dataflow interpreter that processes directed acyclic graphs generated on-the-fly. It is developed for multicore, shared-cache machines, and not for distributed and heterogeneous scenarios. Hence, it cannot be used in the cases studied in this paper. StreamIT (Thies et al. 2002) is a programming language used to design streaming applications as dataflow graphs of data transformation phases, executed on the available resources based on data availability policies. The framework targets clusters of homogeneous nodes. The adopted scheduler does not take into account the status of the network to choose the best mapping of tasks to nodes, a feature strongly required in Fog/In-Situ scenarios. OpenMP (Dagum & Menon 1998), and its new standard 4.0, supports task parallelism (another terminology recently used in place of dataflow model). The runtime targets single-node machines (CPU+GPU nodes). The lack of distributed support makes it currently unsuitable for Fog/In-Situ. S-Net (Penczek et al. 2012) is a mature dataflow model which provides great scalability owing to the stateless design of its nodes. The framework provides a set of low-level optimisations to reduce power consumption in case the underlying resources are underloaded by the current application workload (e.g., using frequency scaling mechanisms of modern's CPUs). Although interesting and useful on commodity server machines, such optimisations are far from being portable to IoT environments.

2.2 Advanced storages for Fog/In-Situ

Fog/In-Situ computing requires highly scalable and available storage distributed services where data shared among tasks are placed and freely obtainable by the system nodes on-demand.

The work in (Jun et al. 2015) presents BlueDBM, a system architecture that employs flash-based storage with in-situ computing capabilities and a low-latency high-throughput inter-controller network. The system is composed of 20 nodes, each having 1TB of flash storage. Storage devices were designed with high-capacity custom flash boards employing FPGAs. They are organised in a network with Near-uniform latency to provide a global address space. BlueDBM allows users to implement custom in-store processing engines and flash cards are designed to expose a set of software interfaces that enables application-specific optimisations in flash accesses: (*i*) a file system interface, (*ii*) a block device driver interface and (*iii*) an accelerator interface. Preliminary experimental results show performance gains of up to an order of magnitude better than clusters equipped with conventional SSDs.

A multipoint approach to address I/O performance bottlenecks in extreme-scale computing is introduced in (Klasky et al. 2011). Authors designed the ADIOS I/O framework following a Service-oriented Architecture (SOA) that takes care of in-situ processing, data staging, data management, application monitoring,

while providing an easy-to-use interface. Moreover, they present ADIOS-BP, a file format that provides resiliency and high performance in extreme-scale systems.

GoogleFS (Ghemawat et al. 2003) is a Linux-based distributed file system developed by Google to provide a reliable and efficient way to store files in clusters of commodity machines. GoogleFS is based on highly scalable mechanisms for accessing large files. The idea is to split files into chunks which are stored in multiple nodes by maintaining the mapping between chunks and nodes in some index structures of the file system. Chunks can be replicated in order to provide data reliability. The architecture is based on the *master-worker* paradigm, with a master in charge of maintaining the metadata of the files and the workers, executing on the different nodes, being responsible for storing the file chunks.

An evolution of GoogleFS is the cross-platform distributed file system adopted by Hadoop (HDFS (Shvachko et al. 2010, The Apache Software Foundation 2016)), which by default uses larger chunks than GoogleFS (128MB vs. 64MB). While GoogleFS supports a multiple-writers multiple-readers model, HDFS is based on the multiple-readers and one-writer model. In HDFS, only the *append* mode is supported to modify files, while GoogleFS allows files to be modified at random positions. This apparent limitation allows HDFS to scale very well in highly distributed scenarios, and to fit perfectly with the MapReduce programming model of which HDFS provides an uniform I/O layer for Hadoop clusters.

Since in this work we focused on enabling Edge/Fog/In-situ support on Sucuri, including scheduling aspects, we did not try to integrate with existing solutions for distributed file system, such as GoogleFS or HDFS. Instead, our solution employs a file catalogue that does not consider chunk division or redundancy of chunks. Studying and implementing such integration is part of our future work.

3 Sucuri

Sucuri (Alves et al. 2014, Silva et al. 2016) is a library written in Python that allows dataflow programming in a higher level than most libraries and runtime environments. It supports transparent execution on multicore clusters, using MPI under the hood. Sucuri main components are `Node`, `Graph`, `Task`, `Worker` and `Scheduler`, described below :

- `Nodes` are objects associated with functions and connected with edges by the programmer (`add_edge` method). Edges describe data dependencies and nodes depict computation that must be performed when dependencies are satisfied.
- A `Graph` object is used as a container, representing the entire dataflow application.
- A `Task` is created by the scheduler once all input operands are available for a certain node. Each task contains the list of input operands and their related node id.
- `Workers` are processes instantiated by Sucuri to execute tasks. When a worker is idle it will request a task to its local scheduler. Once they receive a task they will consult the correspondent node in the graph and call the related function.
- A `Scheduler` is responsible for matching input operands and generating tasks, according to the dataflow firing rule. This means every operand sent by a worker will be stored in a *Matching Unit* until all input operands are available for a certain node, resulting in instantiating a task that is inserted in a *Ready Queue*.

The original Sucuri library (Alves et al. 2014) provided a centralised scheduler, meaning that workers in remote machines will request tasks and deliver results to a dummy local scheduler that will just forward messages to the main scheduler. The Sucuri version used in this paper adopts a distributed scheduler (Silva et al. 2016), meaning that there will be one scheduler instance per machine, each one being responsible for generating tasks for a set of nodes. Moreover, a static mechanism that partitions the graph among Sucuri schedulers is provided.

Figure 1 shows a general view of the Sucuri Architecture. Notice that each machine has its own scheduler. Moreover, the graph is replicated in all machines and each `Scheduler` will have access to a list containing the nodes that are statically mapped to that machine (mappings are represented by colours in the graph). Allocations of tasks to workers are dynamic, following a *First-Come First-Served* policy.

The graph partitioning mechanism used by Distributed Sucuri (Silva et al. 2016) is based on the List Scheduling (LS) algorithm (T. I. Adam & Dickson 1974, Sinnen 2007, H. Topculoglu 2002, Lombardi et al. 2010). The LS version implemented uses a naive priority scheme that has three levels.

Each Graph node has a weight attribute that influences the way the scheduler maps them. They are scheduled taking into account where their incident nodes are. Given two nodes i and j, i is incident to j if and only if, there is at least one edge going from i to j ($i \rightarrow j$).

Allocation starts with source nodes in a circular way over the available cores of each machine, and then it proceeds according to the following rules, also described in detail in (Silva et al. 2016):

1. the scheduler tries to allocate the node in the same worker of the incident node with greater computational cost;
2. it tries to allocate on the same machine of the incident node with greater computational cost;

3. in case incident nodes have the same weight, it allocates in a circular way over workers of incident nodes;

4. if none of the above criteria is matched, it allocates on a remote machine.

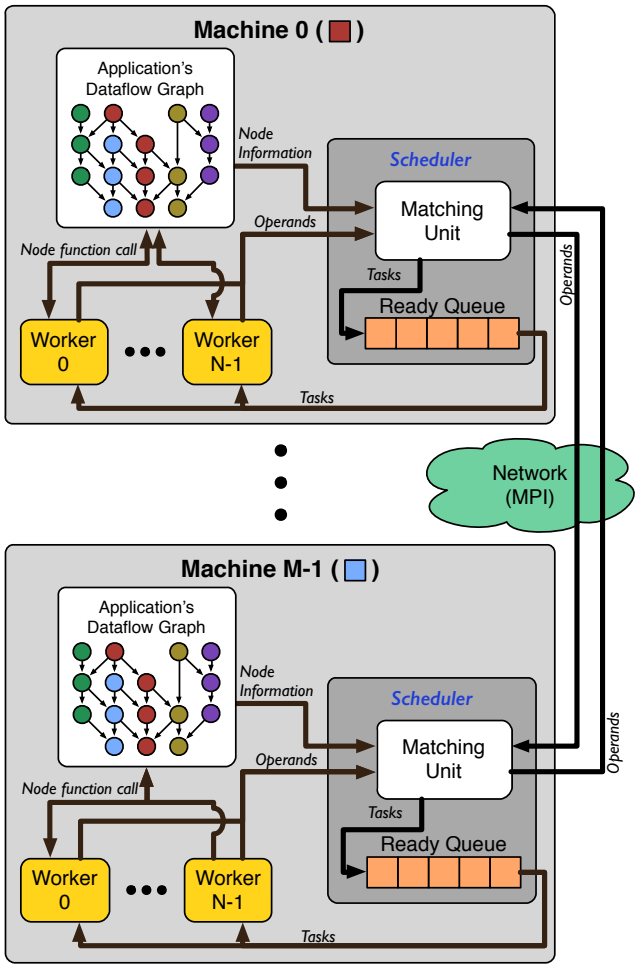


Figure 1 The Distributed Sucuri Architecture (reproduced from (Silva et al. 2016)). The same structure is replicated in each machine. The application dataflow graph is colour-coded, denoting which machine will be responsible from executing tasks generated from each node.

Weights for each node have to be specified by the programmer or by using external profiling tools. In the context of Edge/Fog/In-situ computing, where devices with different processing capabilities are connected in a heterogeneous network, the trade-off between computation and communication should play an important role in scheduling. Moreover, in the context of smart storages, where one can avoid transferring files to perform computation in the disks, it is also important that the runtime and the scheduler know where files are being stored so they can calculate transfer costs more accurately and decide if data should be transferred to the requesting node or processed In-situ. Therefore, Sucuri needs to be adapted in order to consider these issues.

4 Sucuri for Edge, Fog and In-Situ

As the main goal of this work is to turn Sucuri into a Dataflow runtime environment for Edge/Fog/In-situ computing, we propose modifications to Sucuri's static scheduler so that data locality is taken into account when partitioning the dataflow graph. This would allow Sucuri to automatically determine whether it should place nodes closer to where data is sitting (in-situ processing) or it should move data to another machine when its processors would yield higher performance. Notice that the later would result in communication costs, meaning there is a trade-off (communication vs. computation) to be considered. Moreover, a distributed file catalogue was added to Sucuri's scheduler, so it can determine file transfer times in the context of smart storage (with in-situ computing capabilities) and use this information for scheduling purposes.

4.1 Sucuri Environment Setup

In order to make decisions accurately, Sucuri's static scheduler needs to access performance information on the computational platform being used. For that purpose we developed an external tool called SES (Sucuri Environment Setup) that should be executed before running applications on Sucuri. SES contains an `Environment` class that will gather performance information about the computational environment, including network and processing elements performance, and generate a set of configuration files that should be used by Sucuri. SES provides a list of methods to retrieve network information, builds a file catalogue and estimates computation and transfer times. Those methods might be called in bulk or independently via command line interface.

As Sucuri, SES is also implemented in Python and relies on MPI for instantiating remote processes that are used to collect network and performance metric, as well as assembling the File Catalogue with transfer times for each file. SES can be called using the following command line:

```
mpiexec -machinefile <hostfile> -np <n>
        python ses.py <hostfile> -D -B -P
        -f <paths>
```

Where:

- `hostfile` is the list of hosts that will participate in the computation;
- `n` is the number of hosts;
- `-D` informs SES that it should estimate communication latencies between hosts;
- `-B` informs SES that it should estimate communication bandwidth between hosts;
- `-P` informs SES that it should estimate computation performance of each hosts;

- `-f <paths>` informs SES that it should build a *File Catalogue* using files indicated in a list (`paths`);

4.1.1 Network information gathering with SES

Latency and bandwidth have a huge effect on file transfer times in a network storage. Moreover, due to factors like reception and transmission overhead or costs related to the communication protocol itself (such as congestion control on TCP), the application network layer can only utilise a fraction of the available bandwidth. Considering that one of our goals in this configuration phase is to provide a good estimation on data transfer time between hosts, we need to know the amount of useful data effectively received by the application in a period of time (goodput), as well as latency and bandwidth between each pair of machines (or Edge/Fog/In-situ devices). For this purpose SES provides the following methods: `getLatencies`, `getBandwidth` and `getGoodput`.

The `getLatencies` method measures communication latencies between each pair of machines (or Edge/Fog/In-situ devices) using Linux `ping` command and builds an adjacency matrix.

The `getBandwidth` method measures communication bandwidth between each pair of machines (or Edge/Fog/In-situ devices) using Linux `iperf` command and builds a bandwidth matrix. However, since determining bandwidth can be more complex and costly, and given that `iperf` utilises a client/server approach, `getBandwidth` will only fill in the upper triangular matrix. Moreover, measurements are taken in multiple steps, where each step comprise of a group of measures taken in parallel without repeating the source or destination machines. This is done in order to minimise the interference of having a machine communicating simultaneously with two other machines in a given step.

The `getGoodput` method estimates the amount of useful data that can be effectively received by an application in a period of time. Some previous works model TCP performance and could be employed for determining the goodput, such as (Mathis et al. 1997, Altman et al. 2005, Fortin-Parisi & Sericola 2004). These works, however, are too restrictive or depend on parameters that cannot be trivially obtained at the application layer, unless there is a structural analysis of network packets. For this reason, we decided to employ an empirical approach.

Intuitively, file transfer time (T) will be proportional to both the file size (s) and a goodput constant (g). Therefore, we could transfer a file between hosts, measure the time spent on the process and obtain the value of g using the following equation:

$$g = \frac{T}{s} \tag{1}$$

It is important to mention that the value of g must be calculated based on transfer of files that are not too small, when compared to the bandwidth. Using small files would result in measurement distortion, given that overheads would tend to dominate transfer times. To overcome this issue a second constant o was added to Equation (1). The value of o can be obtained transferring a very small file (in the order of bytes), where $o = T$, given that s is too small. Then we obtain the following equation for file transfer time:

$$T = s \times g + o \tag{2}$$

This approach is not only simple, but also does not require any parameters that depend on the implementation peculiarities of protocols, thus making it flexible enough to be adopted in different environments. Moreover, we do not need extreme precision in our estimation, since those values would just help Sucuri's scheduler in its decisions. Results presented in Section 5 validate our approach.

4.1.2 Building the File Catalogue with SES

Our solution requires users to register paths for all files that are going to be used in their applications. This is necessary because Sucuri scheduler needs to know the physical location of files, so it can calculate file transfer times. File registration can be done either by a configuration file passed to SES or by calling the `registerFile` method in SES. Sucuri Environment Setup will discover the size and host for each file.

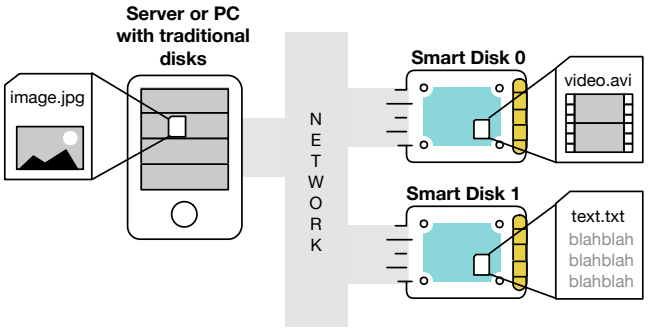
Using all collected information, and using Equation (2), SES estimates transfer time for each file to all possible hosts in the system, using the `getTransferTimes` method. Then, it generates a *File Catalogue* that will be used by Sucuri's static scheduler.

Figure 2 provides a full example on how the user could generate a file catalogue that keeps track of three files (`video.avi`, `image.jpg` and `text.txt`). Figure 2(a) shows that files are distributed in two smart disks and a traditional server. Figure 2(b) shows the use of `registerFile` method to inform which files are going to be on the catalogue. Users can also pass a file to SES, containing a list of paths to be registered. Finally, Figure 2(c) shows the File Catalogue containing the paths of all files and transfer times to all hosts in the system.

In this work, we are not yet integrating with existing solutions for distributed storage, such as HDFS or GFS. In the future, Sucuri file catalogue could be eliminated and we could query those services to determine file location. However, in HDFS and GFS, files are split into blocks which would require us to place the computation closer to most blocks or closer to the blocks most likely to be accessed. Redundancy is also implemented by those systems and Sucuri could take advantage of that.

4.1.3 Estimating Computing Performance with SES

The `getPerformance` method allows SES to estimate computing performance for machine (or Edge/Fog/In-situ device). This is done by executing a set of synthetic benchmarks with different complexities ($\mathcal{O}$ notation). We



(a) File Location

```
ses.registerFile("/path1/image.jpg")
ses.registerFile("/path2/video.avi")
ses.registerFile("/path3/text.txt")
```

(b) File Registration

File Path	Transfer Times		
	Server	Disk 0	Disk 1
/path1/image.jpg	0	400.5	280.7
/path2/video.avi	600.3	0	200.5
/path3/text.txt	300.34	110.4	0

(c) File Catalogue

Figure 2 File Catalogue in a smart storage environment

have two applications for each complexity class and two input sets per application with different sizes. For each application we calculate the time corresponding to one computational step (t_1) by dividing the execution time (t) by the application known complexity:

$$t_1 = \frac{t}{complexity} \tag{3}$$

For example, in an application with $complexity = \mathcal{O}(n \log n)$, where n is the input size, substituting on (3) we would have:

$$t_1 = \frac{t}{n \log n} \tag{4}$$

This way we can estimate later the time an algorithm will take to run on each machine, given that its complexity and size of input are know. Currently, SES includes benchmarks with complexity of $\mathcal{O}(n)$, $\mathcal{O}(\log n)$, $\mathcal{O}(n \log n)$ and $\mathcal{O}(n^2)$. After all measurements a *Computational Cost Matrix* will be produced, where each line will represent a machine and each column will represent t_1 for algorithms with different complexities (4 columns), taken from the mean of t_1 of each machine.

It should be noted that the $\mathcal{O}$ notation here was used just as an illustration, because we are not using this information (if algorithm is $\mathcal{O}$ or Θ, for example) directly in t_1 calculations.

4.2 Edge/Fog/In-situ static scheduling

In the previous work of (Carvalho et al. 2017), the original Sucuri scheduler (discussed in Section 3) was modified to always place computation where data is sitting. Nodes of the dataflow graph that manipulate files would be scheduled to run on the processors of the smart disk that stores those files. If, according to the static scheduler, the disk processor would be busy running another node, then the decision would be to choose the closest machine (or device).

In this work, we took a step forward and included performance in the decision process. Notice that smart disks would probably have a slower processor than a traditional server. On the other hand, executing a task on the disk processor would save us file transfer time (through network). Information generated by SES is passed to Sucuri in the form of configuration files so that Sucuri static scheduler can partition the graph to maximise performance with locality awareness.

Our scheduler will allocate each node of the dataflow graph on the machine (or device) which has the lowest overall cost (C), based on the computational cost (C_{comp}) and communication cost (C_{comm}):

$$C = C_{comp} + C_{comm} \tag{5}$$

Notice that C_{comm} is basically the file transfer time T (available in the *File Catalogue*), while C_{comp} is based on the profile information for an algorithm of same complexity (determined by SES) and the file size (s):

$$C_{comp} = t_1 \times s \tag{6}$$

The user needs to inform the complexity of the algorithm running on each dataflow node so that the scheduler can estimate the execution time based on the right profile information from SES. The method `set_complexity` should be called on the `Node` object for that purpose. Then, Sucuri scheduler will build an array for each dataflow node n that manipulates a file where each element will hold C_i, which is the cost C of running n on host i. The scheduler will select host m with minimum cost C_{min} for allocating n.

It is important to mention that, since this is a static scheduling policy, it can be performed once for several executions of the application. Mappings are saved to a file where each line represents a node in the order they were added to the graph and each number on the line is the machine where that node was mapped to. Moreover, as for the local schedulers running on each machine, they still adopt a pool of tasks, which means it can perform load balancing in an on-demand basis.

5 Experiments and Results

Our experimental environment consisted of a PC equipped with an Intel® Core™ i5-3210M CPU (2.50Ghz quad core), 4GB of memory, running a 3.10 Linux kernel, and a Parallella board (Parallella 2014), equipped with a Xilinx Zynq Z7010 (ARM Cortex A9 Dual core + FPGA) (Xilinx 2017), an Epiphany 16-core RISC processor, 1GB of memory and running a 4.6 Linux kernel. The devices were connected by Gigabit Ethernet.

At the Parallella board we used an SD card as the storage medium for all input files used in our experiments.

Since the Parallella board had lower processing power than the PC, we evaluated the proposed solution conducting a set of experiments with a set of well-known algorithms with different costs (complexities):

- **Search** performs a sequential search in a text file counting all occurrences of a certain character. Search complexity is $\mathcal{O}(n)$.
- **Search 2x** performs the same operation done by **Search** on 2 files of the same size. This application just replicates the Sucuri node that performs the search operation to take advantage of different cores available either in the Parallella board or the PC. Search 2x complexity is also $\mathcal{O}(n)$.
- **Filter** finds numbers in lines of an input file, using a regular expression, and adds them up. Each line has 24 characters. Considering that regular expression matching on a string with m characters has $\mathcal{O}(m)$ complexity and that we are performing that operation on a list of n strings, complexity would be $\mathcal{O}(n \times m)$. However, since $n \gg m$ we could assume a complexity of $\mathcal{O}(n)$. Note that Filter has a constant ($\mathcal{O}(1)$ time) that is considerably larger than Search. This would be a good opportunity for evaluating our computation time estimation mechanism.
- **HeapSort** orders a binary file containing 8-bit integers using the Heapsort algorithm ($\mathcal{O}(n \log n)$ complexity).
- **MergeSort** orders a binary file containing 8-bit integers using the Mergesort algorithm ($\mathcal{O}(n \log n)$ complexity).
- **SelectionSort** orders a binary file containing 8-bit integers using the SelectionSort algorithm ($\mathcal{O}(n^2)$ complexity).
- **InsertionSort** orders a binary file containing 8-bit integers using the InsertionSort algorithm ($\mathcal{O}(n^2)$ complexity).

Complexities were indicated here in $\mathcal{O}$ notation because this is the common notation for these known algorithms. In algorithms with Θ values better than $\mathcal{O}$ ones, Θ could be used for better precision.

All benchmarks were implemented in Python with no optimisation efforts to avoid Python interpretation overheads, such as using pre-compiled C functions as dynamic libraries that could be invoked from Python code. Since file size plays an important role in our study, all experiments present results for different file sizes. For Search and Search 2x we used input files of 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, and 476MiB. For Filter, we executed with files of 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB and 48MiB. For HeapSort and MergeSort, files of 256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB were used. For SelectionSort and InsertionSort, files of 256 bytes, 4KiB, 8KiB, 48KiB and 256KiB were used. The maximum file size evaluated for each application was chosen so that the application could be stressed in the context of in-situ computing.

In the first set of experiments, we enforced In-situ execution, meaning that all Sucuri nodes that manipulate input files should execute on the ARM cores of the Parallella board. Then, we activate the proposed scheduler to evaluate if it can successfully prevent outsourcing execution to the Parallella and avoid performance losses.

Figure 3 shows the results for Search and Search 2x applications. The y-axis show speedups based on PC execution times, where data is always copied from the Parallella board. Notice that enabling in-situ provided speedup gains, regardless of file sizes. Moreover, the Sucuri scheduler always took the right decisions (to use the board). Therefore, results of enforced in-situ are the same of using our scheduler to decide.

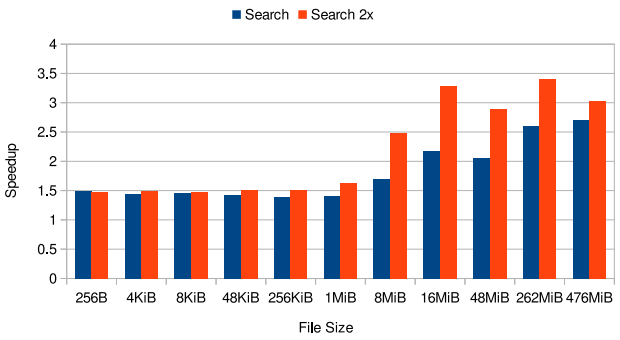


Figure 3 Speedups for the search algorithm when using the proposed scheduler for one and two (Search 2x) input files. The x axis shows shows file sizes, while the y axis shows the speedup over a non-in-situ (original) scenario. Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, and 476MiB).

Figure 4 shows the results for the Filter application. The y-axis shows speedups based on PC execution times, where data is always copied from the Parallella board. Notice that Sucuri scheduler always decided to enable in-situ, which was not the correct decision for input files larger than 1MiB. Although Filter complexity is $\mathcal{O}(n)$, the base time (t_1) is much larger than the ones in $\mathcal{O}(n)$ applications used in SES and in Search application. This suggests that our scheduler could accept such constant as an input, which will be left to future work.

Figure 5 shows the results for sort applications. The y-axis shows speedups based on PC execution times, where data is always copied from the Parallella board. In Figure 5(a), in-situ execution is enforced and, in Figure 5(b), we are relying on our scheduler

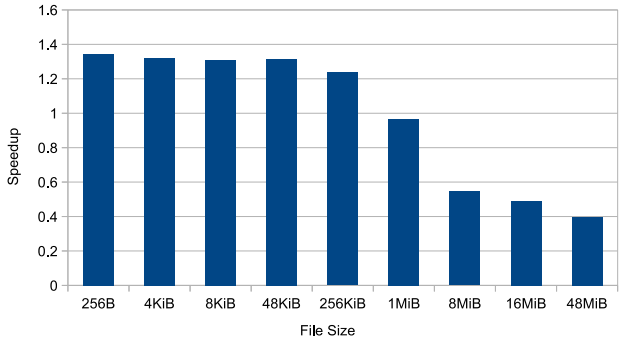
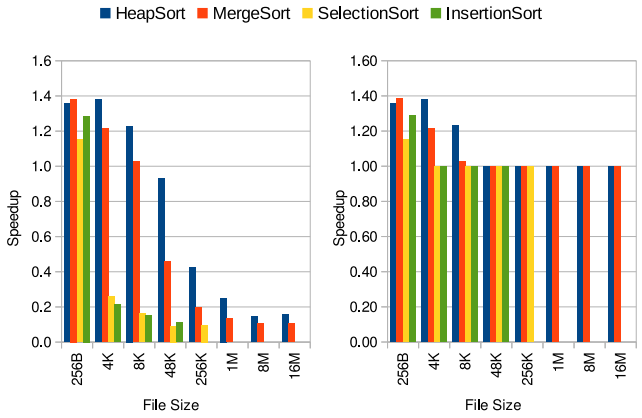


Figure 4 Speedups for the Filter application when using the proposed scheduler. The x axis shows shows file sizes, while the y axis shows the speedup over a non-in-situ (original) scenario. Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB and 48MiB).



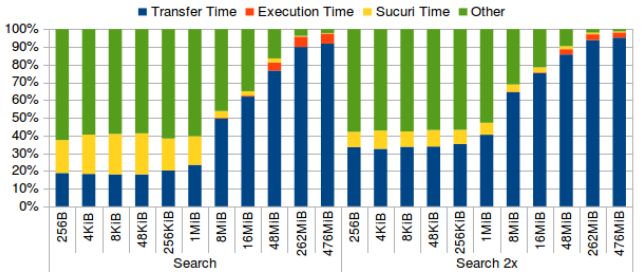
(a) Enforced In-situ (b) Scheduled In-situ

Figure 5 Speedups for the different sorting algorithms when enforcing in-situ and when using the proposed scheduler. The x axis shows file sizes, while the y axis shows the speedup over a non-in-situ (original) scenario. Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB).

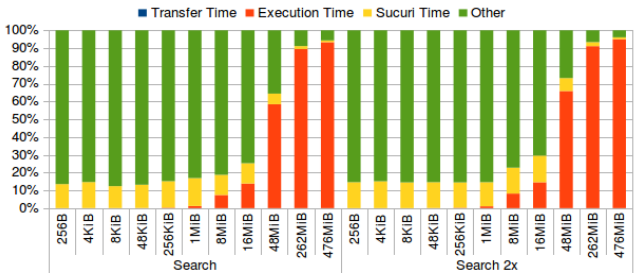
to make that decision. Notice that in-situ execution is only good for smaller files. Moreover, applications with greater complexities provide the worse results. Since the Parallella board has much less resources, such as a smaller memory bus, cache size and slower processor when compared to the PC, this would be expected. Another important factor is that we limited the number of cores for both processors to two. The Intel processor can use the other cores to execute the Operating System, while the ARM processor would need to share its two cores with the Operating System. Nevertheless, our proposed scheduler made the right choices and prevented performance losses by disabling in-situ execution for larger files, since transferring files would yield higher performance than running on a slower processor. In-situ execution of HeapSort and MergeSort yielded speedups of up to 1.4 for files with up to 8KiB. SelectionSort and InsertionSort only provided performance gains for 256-byte files.

In order to evaluate the potential performance gains of our approach, we measured the percentage of time spent on file transfers, application execution, Sucuri overheads and other overheads (such as MPI initialisation). That was done for both the original applications (in-situ disabled) and the ones that use in-situ execution guided by our scheduler. Figures 6 and 8 provide results for search and sort applications respectively.

In Figure 6, overheads (Sucuri and other) are predominant for small files, since total time for those cases is too small (in the order of 2s). For larger files, transfer times dominate the execution for non-in-situ configurations (Figure 6(a)). On the other hand, transfer time is completely eliminated in all scenarios of Figure 6(b), since the scheduler is always deciding to enable in-situ. Actually, having file transfer as a dominating aspect of this application is exactly what makes it a good candidate for in-situ execution, since the lower



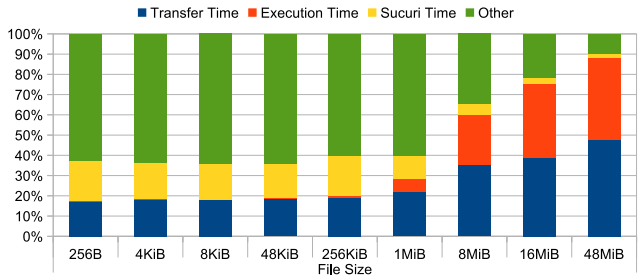
(a) Original



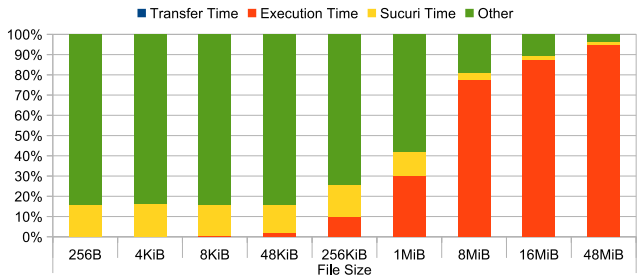
(b) In-situ (scheduler)

Figure 6 Time distribution for the Search application with one and two input files (2x). Results are provided for both the original and In-situ scenarios. The x axis shows file sizes, while the y axis shows % of time spent in transferring files, executing the algorithm, Sucuri overhead and other (such as MPI initialisation costs). Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB, 48MiB, 262MiB, and 476MiB).

performance of the smart disk processing cores would not be a big obstacle.



(a) Original



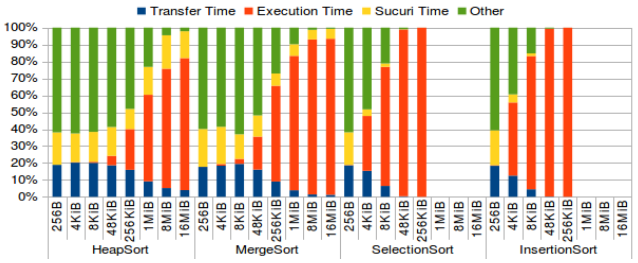
(b) In-situ (scheduler)

Figure 7 Time distribution for the Filter application. Results are provided for both the original and In-situ scenarios. The x axis shows file sizes, while the y axis shows % of time spent in transferring files, executing the algorithm, Sucuri overhead and other (such as MPI initialisation costs). Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB, 16MiB and 48MiB.

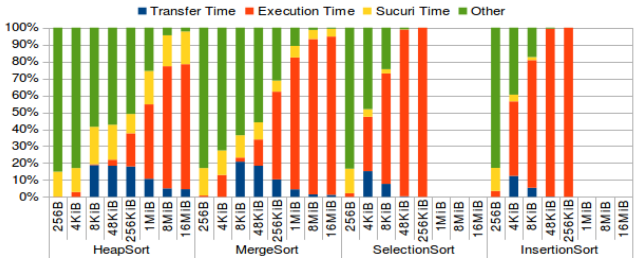
In Figure 7, overheads (Sucuri and other) are also predominant for small files, since total time for those cases is too small (in the order of 2s). However, for larger files, transfer times and execution times are well balanced, suggesting there could be gains with in-situ computing (Figure 7(a)). Slowdowns happened because the Parallella board is way less powerful than the PC used in our experiments .

In Figure 8, it is possible to observe a different scenario. Overheads (Sucuri and other) dominate only for really small files and execution time dominate for larger ones, specially for SelectionSort and InsertionSort. In this case, faster in-situ processors would be required, in order to yield performance gains by avoiding such insignificant file transfers.

To evaluate our approach on the context of Edge/Fog applications, where network can present higher latencies and packet loss than in a local networks, we also conducted a set of experiments varying those parameters. We expect to improve gain on in-situ computing in those scenarios, since transfer costs would be higher. Search and Search 2x applications were not included in those experiments because they already present good results in a local network. Latency and



(a) Original



(b) In-situ (scheduler)

Figure 8 Time distribution for the different sorting algorithms on the original and In-situ scenarios. The x axis shows shows file sizes, while the y axis shows % of time spent in transferring files, executing the algorithm, Sucuri overhead and other (such as MPI initialisation costs). Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB).

packet loss rate were enforced using *netem*, a network emulation functionality provided by the Linux kernel (Linux Foundation Wiki 2017). We simulated delays of 100ms, 200ms and 300ms, with 10ms variation using a normal distribution, and packet losses of 5% and 15%. Delays of this order of magnitude could be found when data is being stored on a distant server. As for the packet loss, those values are more common on a scenario where there are sensors communicating through a wireless connection which might suffer from electromagnetic interference. Notice that those characteristics are very common in IoT applications that employ smart sensors using wireless connections to send data to the cloud.

Figure 9 shows the results for latency experiments, using Sucuri scheduler. Notice that, as latency increases, in-situ execution becomes profitable for larger files, specially for HeapSort and MergeSort (speedups for files with up to 48KiB). SelectionSort and InsertionSort did not present significant improvements. Moreover, there is an upper bound trend of about 1.6 speedup, possibly because the application is being able to stress one ARM core of the Parallella board even for smaller file sizes.

Figure 10 shows the results for packet loss experiments, using Sucuri scheduler. Notice that, as packet loss rate increases, in-situ execution tends to become profitable for larger files. However, since packet loss would also affect packets of Sucuri runtime, we got high standard deviations for packet loss rate of 15%.

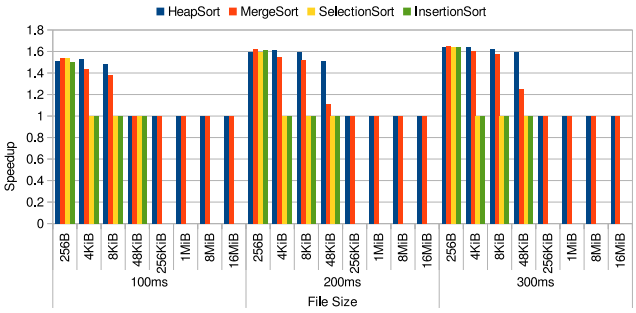


Figure 9 Speedups for the different sorting algorithms when enforcing different network latencies and using the proposed scheduler. The x axis shows file sizes and latencies, while the y axis shows the speedup over a non-in-situ (original) scenario. Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB).

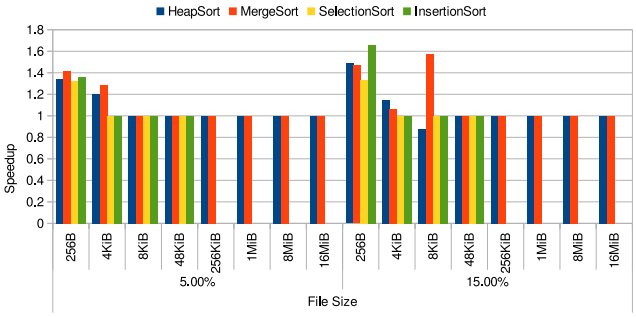


Figure 10 Speedups for the different sorting algorithms when enforcing different network packet loss rates and using the proposed scheduler. The x axis shows file sizes and packet loss rates, while the y axis shows the speedup over a non-in-situ (original) scenario. Each bar provides the results for the input files size used (256 bytes, 4KiB, 8KiB, 48KiB, 256KiB, 1MiB, 8MiB and 16MiB).

Besides the low processing power of the ARMs, which removed part of the gains we could achieve with data locality, it is important noticing that our experiments only covered applications that read data from the files. In a storage scenario, for example, in which applications would also update files, the transfer times from the regular computer back to the board should be taken into account as well, resulting in potential greater speedups.

6 Discussion and Future Work

In this work, we propose to transform Sucuri, a Dataflow programming library for Python, making it capable of executing in-situ processing in heterogeneous environments. The library was already versatile to allow transparently execution on clusters of multicores, and now Sucuri also deals with data locality in a straightforward way. Sucuri scheduler was modified to consider file transfer times (through network) and computation times to decide whether it should outsource execution of nodes of the application dataflow graph to smart disks that store input data.

Sucuri Environment Setup (SES) tool was developed to allow developers to estimate computing power, bandwidth, latency and transfer times of different machines. SES also aids to build a File Catalogue that is used by Sucuri scheduler.

Experiments with a set of benchmarks with different complexities show performance gains in cases where data transfers are more expensive than computational costs and also when network latency or packet loss are high. Also, in every case but one, the scheduler took the right performance decision about where to process the work: in-situ or transferring the input to the more powerful machine. This demonstrates Sucuri's ability to orchestrate dataflow parallelism in multicores, clusters and in-situ environments transparently and efficiently. Moreover, it was possible to identify limitations pertaining the equipment used to represent the storage device.

This work opens a set of possible future research, such as experimenting with different benchmarks, evaluating different devices, and performing experiments to measure energy consumption as another possible benefit of in-situ computing. Moreover, in order to increase performance, we need to fully explore the resources of the Parallella board, including the FPGA and Epiphany cores. It was also detected that in some cases we could benefit from having the complexity constants in the performance calculations, a good point for future work. Finally, it might be useful to experiment with much larger datasets and applications that generate output data, to stress the library and seek for possible bottlenecks we could fix.

We are also working on implementing a dynamic scheduling mechanism for Sucuri in the context of Edge/Fog/In-situ computing. The scheduler could invoke SES to dynamically update system profile

information. Moreover, we intend to integrate it with some distributed file system inspired in the GFS (Ghemawat et al. 2003), like the open source Hadoop Distributed File System (Shvachko et al. 2010, The Apache Software Foundation 2016). They are mature and specialised file systems for exploiting data locality with large files and redundancy, thus our approach can benefit from their features and can allow developers of Sucuri to concentrate on the dataflow aspects.

References

Altman, E., Avrachenkov, K. & Barakat, C. (2005), 'A Stochastic Model of TCP/IP With Stationary Random Losses', *IEEE/ACM Transactions on Networking* **13**(2), 356–369.

Alves, T. A., Goldstein, B. F., França, F. M. & Marzulo, L. A. (2014), A Minimalistic Dataflow Programming Library for Python, *in* '2014 International Symposium on Computer Architecture and High Performance Computing Workshop', IEEE, pp. 96–101.

Alves, T. A. O., Marzulo, L. A. J., França, F. M. G. & Costa, V. S. (2011), 'Trebuchet: Exploring tlp with dataflow virtualisation', *Int. J. High Perform. Syst. Archit.* **3**(2/3), 137–148.

Buono, D., Danelutto, M., De Matteis, T., Mencagli, G. & Torquati, M. (2014), A lightweight run-time support for fast dense linear algebra on multi-core, *in* 'Proc. of the 12th International Conference on Parallel and Distributed Computing and Networks (PDCN 2014). IASTED, ACTA press'.

Carvalho, C. B. G., Ferreira, V. C., França, F. M. G., Bentes, C., Alves, T. A. O., Sena, A. C. & Marzulo, L. A. J. (2017), Towards a dataflow runtime environment for edge, fog and in-situ computing, *in* '2017 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)', pp. 115–120.

Caulfield, A., Chung, E., Putnam, A., Angepat, H., Fowers, J., Haselman, M., Heil, S., Humphrey, M., Kaur, P., Kim, J.-Y., Lo, D., Massengill, T., Ovtcharov, K., Papamichael, M., Woods, L., Lanka, S., Chiou, D. & Burger, D. (2016), A cloud-scale acceleration architecture, *in* 'Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture', IEEE Computer Society.

Dagum, L. & Menon, R. (1998), 'Openmp: An industry-standard api for shared-memory programming', *IEEE Comput. Sci. Eng.* **5**(1), 46–55.

Duro, F. R., Blas, J. G., Isaila, F., Wozniak, J., Carretero, J. & Ross, R. (2014), 'Exploiting data locality in Swift / T workflows using Hercules', *Nesus Workshop* **I**(1), 71–76.

Fitzpatrick, B. (2004), 'Distributed caching with memcached', *Linux J.* **2004**(124), 5–.

Fortin-Parisi, S. & Sericola, B. (2004), 'A Markov model of TCP throughput, goodput and slow start', *Performance Evaluation* **58**(2-3), 89–108.

Ghemawat, S., Gobioff, H. & Leung, S.-t. (2003), The Google file system, *in* 'Proceedings of the nineteenth ACM symposium on Operating systems principles - SOSP '03', ACM Press, New York, New York, USA, pp. 29–43.

Giang, N. K., Blackstock, M., Lea, R. & Leung, V. C. M. (2015), Developing iot applications in the fog: A distributed dataflow approach, *in* '2015 5th International Conference on the Internet of Things (IOT)', pp. 155–162.

H. Topculoglu, S. Hariri, M. W. (2002), Performance-effective and low-complexity task scheduling for heterogeneous computing, *in* 'IEEE Trans. Parallel Distrib. Systems 13 (3)', pp. 260–274.

Jalaparti, V., Bodik, P., Menache, I., Rao, S., Makarychev, K. & Caesar, M. (2015), 'Network-Aware Scheduling for Data-Parallel Jobs', *ACM SIGCOMM Computer Communication Review* **45**(5), 407–420.

Johnston, W. M., Hanna, J. R. P. & Millar, R. J. (2004), 'Advances in dataflow programming languages', *ACM Computing Surveys* **36**(1), 1–34.

Jun, S. W., Liu, M., Lee, S., Hicks, J., Ankcorn, J., King, M., Xu, S. & Arvind (2015), BlueDBM: An appliance for Big Data analytics, *in* '2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)', pp. 1–13.

Juniper (2018), 'Juniper Networks Advances Application Performance and Acceleration with New Compact Compute-Integrated Network Switch'. Available at `https://www.maxeler.com/juniper-switch/`. Accessed: Feb 1, 2018.

Kim, J., Abbasi, H., Chacn, L., Docan, C., Klasky, S., Liu, Q., Podhorszki, N., Shoshani, A. & Wu, K. (2011), Parallel in situ indexing for data-intensive computing, *in* '2011 IEEE Symposium on Large Data Analysis and Visualization', pp. 65–72.

Klasky, S., Abbasi, H., Logan, J., Parashar, M., Schwan, K., Shoshani, A., Wolf, M., Ahern, S., Altintas, I., Bethel, W. et al. (2011), 'In situ data processing for extreme-scale computing', *Scientific Discovery through Advanced Computing Program (SciDAC11)*.

Linux Foundation Wiki (2017), 'netem'. Available at `https://wiki.linuxfoundation.org/networking/netem`. Accessed: 28-Aug-2017.

Lombardi, M., Milano, M., RUGGIERO, M. et al. (2010), Stochastic allocation and scheduling for conditional task graphs in multi-processor systems-on-chip, *in* 'Journal of Scheduling doi: 10.1007/s10951-010-0184-y.', pp. 315–345.

Marzulo, L. A., Alves, T. A., França, F. M. & Costa, V. S. (2014), 'Couillard: Parallel programming via coarse-grained data-flow compilation', *Parallel Computing* **40**(10), 661 – 680.

Matheou, G. & Evripidou, P. (2016), FREDDO: an efficient framework for runtime execution of data-driven objects, *in* 'Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)', Las Vegas, pp. 265–273.

Mathis, M., Semke, J., Mahdavi, J. & Ott, T. (1997), 'The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm', *ACM SIGCOMM Computer Communication Review* **27**(3), 67–82.

NGD Systems (2017), 'NGD Systems announces availability of industry's first Computational Storage'. Available at `http://www.prnewswire.com/news-releases/ngd-systems-announces-availability-of-industrys-first-computational-storage-300493319.html`. Accessed: Feb 1, 2018.

OpenFog Consortium Architecture Working Group (2016), 'OpenFog Architecture Overview'. Available at `http://www.openfogconsortium.org/wp-content/uploads/OpenFog-Architecture-Overview-WP-2-2016.pdf`. Accessed: Sep 7, 2017.

Parallella (2014), 'Parallella-1.x reference manual'. Available at `http://www.parallella.org/docs/parallella_manual.pdf`. Last accessed on February 1, 2018.

Penczek, F., Cheng, W., Grelck, C., Kirner, R., Scheuermann, B. & Shafarenko, A. (2012), A data-flow based coordination approach to concurrent software engineering, *in* '2012 Data-Flow Execution Models for Extreme Scale Computing', pp. 36–43.

Sena, A. C., Vaz, E. S., França, F. M. G., Marzulo, L. A. J. & Alves, T. A. O. (2015), Graph templates for dataflow programming, *in* '2015 International Symposium on Computer Architecture and High Performance Computing Workshop (SBAC-PADW)', pp. 91–96.

Shi, W., Cao, J., Zhang, Q., Li, Y. & Xu, L. (2016), 'Edge computing: Vision and challenges', *IEEE Internet of Things Journal* **3**(5), 637–646.

Shvachko, K., Kuang, H., Radia, S. & Chansler, R. (2010), The hadoop distributed file system, *in* 'Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)', MSST '10, IEEE Computer Society, Washington, DC, USA, pp. 1–10.

Silva, R. J., Goldstein, B., Santiago, L., Sena, A. C., Marzulo, L. A., Alves, T. A. & França, F. M. (2016), Task Scheduling in Sucuri Dataflow Library, *in* '2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)', Vol. 1, IEEE, pp. 37–42.

Sinnen, O. (2007), *Task Scheduling for Parallel Systems (Wiley Series on Parallel and Distributed Computing)*, Wiley-Interscience.

T. I. Adam, K. M. C. & Dickson, J. R. (1974), A comparison of list schedules for parallel processing systems, *in* 'Commun. ACM, vol. 17, no.12', pp. 685–690.

TBB (2014), 'Tbb flowgraph'. Available at `http://www.threadingbuildingblocks.org/docs/help/reference/flow_graph.htm`. Last accessed on August 8, 2014.

The Apache Software Foundation (2016), 'HDFS Users Guide'. Available at `http://hadoop.apache.org/docs/r2.7.3/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html`. Accessed: 29-Aug-2017.

Thies, W., Karczmarek, M. & Amarasinghe, S. P. (2002), Streamit: A language for streaming applications, *in* 'Proceedings of the 11th International Conference on Compiler Construction', CC '02, Springer-Verlag, London, UK, UK, pp. 179–196.

Wilde, M., Hategan, M., Wozniak, J. M., Clifford, B., Katz, D. S. & Foster, I. (2011), 'Swift: A language for distributed parallel scripting', *Parallel Computing* **37**(9), 633 – 652.

Wozniak, J., Armstrong, T., Wilde, M., Katz, D., Lusk, E. & Foster, I. (2013), 'Swift/t: Large-scale application composition via distributed-memory dataflow processing', pp. 95–102.

Xilinx (2017), 'Zynq-7000 all-programmable technical reference manual'. Available at `https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf`. Accessed: Sep 7, 2017.