applied sciences

MDPI

Technical Note

CSS: Container Resource Manager Using System Call Pattern for Scientific Workflow

Chunggeon Song [1], Heonchang Yu [1] and Eunyoung Lee [2,*]

1 Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea
2 Department of Computer Science, Dongduk Women's University, Seoul 02748, Korea
* Correspondence: elee@dongduk.ac.kr; Tel.: +82-2-940-4588

Abstract: Multiple containers running scientific workflows in SMP-based high-performance computers generate some bottlenecks due to workload flexibility. To improve system resource utilization by minimizing these bottlenecks, vertical resource management is required to determine an appropriate resource usage policy according to the resource usage type of the container. However, the traditional methods have additional overhead for collecting monitoring metrics, and the structure of the resource manager is complex. In this paper, in order to compensate for these shortcomings, we propose CSS, a dynamic resource manager utilizing system call data collected for security purposes. The CSS utilizes the SBCC algorithm, which uses the number of futex system calls as a heuristic measure to determine the number of IO-intensive workload occurrences. In addition, the CTBRA algorithm is used to determine the range of resources to be allocated for each container and to perform actual resource allocation. We implemented a prototype of CSS and conducted experiments on NPB to analyze the performance of CSS with various types of large-scale tasks of a scientific workflow. As a result of the experiment, it showed a performance improvement of up to 7% compared with the environment where Linux cgroups were not applied. In addition, CANU performance analysis was performed to verify the effectiveness of applications used in the real world, and performance improvement of up to 4.5% was shown.

Keywords: container; dynamic resource manager; scientific workflow; system call

check for updates

Citation: Song, C.; Yu, H.; Lee, E. CSS: Container Resource Manager Using System Call Pattern for Scientific Workflow. *Appl. Sci.* **2022**, *12*, 8228. https://doi.org/10.3390/app12168228

Academic Editors: Vassilios V. Dimakopoulos and Spiridoula V. Margariti

Received: 22 June 2022
Accepted: 10 August 2022
Published: 17 August 2022

1. Introduction

Container technology helps to agilely deploy an independent lightweight execution environment for tasks dependent on a certain library (lib) and a certain command (bin). There is a growing demand for applying these container techniques to scientific workflow applications. In a scientific workflow, various types of tasks have dependencies on each other, and the tasks are processed through several stages. Since the tasks in each stage differ in the types and requirements of computing resources used, the competition rate for preempting a specific resource may increase, or excessive resources may be preempted. This condition causes system performance degradation. Accordingly, resource management of multiple containers running concurrently within a single server is an important factor in determining overall system performance. Accordingly, resource management of multiple containers running concurrently within a single server is an important factor in determining overall system performance, and various related studies have been proposed [1–3].

One of the major challenges in container resource management research is the problem of minimizing bottlenecks arising from competition for resources in a high-performance system in which multiple containers operate. Existing studies on container resource management have disadvantages such as causing additional overhead for collecting monitoring metrics or complicating the system structure [1–5]. In addition, existing studies have focused on the scheduling technique that selects cluster nodes at the time of container deployment but neglected resource competition that occurs after deployment [4–6].

Our study began with the idea of using the system call pattern that occurs in tasks executed in the container as heuristic information for the resource management technique. For the security of the operating system kernel, all the recent container runtime systems include a function to collect system calls [7]. Through the runtime systems' filtering technique, only the authorized system calls are allowed to proceed after comparison with the white list of system calls. We focused on minimizing the overhead generated during collecting the resource usage of containers and simplifying the internal structure of the container resource manager. We also designed a resource management technique for minimizing resource competition that occurs after container distribution through vertical resource management using Linux cgroups (*cgroups*) [8].

As a result of these ideas, we propose CSS (Container resource manager using System call pattern for Scientific workflow) to perform lightweight resource management for Singularity containers based on the MAPE-K model in a high-performance computing system. We compared CSS with the default *cgroups* policy of Singularity; we conducted experiments and analyzed the result to confirm the degree of performance improvement. The main contributions of our research are as follows:

- We developed a technique, SBCC, to classify the types of containers' resource usage by using the number of futex system calls of tasks executed in the Singularity container runtime.
- We developed a technique, CTBRA, that allocates minimum resources to containers performing IO-intensive tasks and distributes isolated resources equally to containers performing CPU-intensive tasks.
- We developed CSS, a prototype research manager that implements the proposed dynamic resource management technique and demonstrated its effectiveness by performing experiments with NPB (NAS Parallel Benchmark) and CANU framework work.

We will organize the rest of the paper as follows. Section 2 describes the implementation that motivated us to come up with the idea of using system call information for container resource management. Section 3 describes the structure, components, and core algorithm of the proposed CSS. The implementation method of CSS, performance experiments, and analysis of the results are given in Section 4. Section 5 introduces related studies, and, finally, we summarize the contribution of the study and discuss the future research direction in Section 6.

2. Motivation

2.1. System Call Pattern

In order to assess the feasibility of the idea of using system call information for container resource management, a typical scientific workflow task was repeatedly executed, and an experiment was conducted to determine whether a consistent system call pattern appears. We selected the Linux kernel build task for observation. System call data generated in the OS kernel were collected using the Strace tool while performing the Linux kernel build task. The task was executed in the Singularity container runtime environment [9]. The task was run five times, and the same result was obtained. Figure 1 shows the name and number of system calls that were generated by the Linux kernel build task.

As a result of execution, we observed that a consistent system call pattern is shown in the Linux kernel build task, and some system calls occur much higher than other system calls. In the next step, an experiment was conducted to check whether it was possible to collect the information on the type of resources in use from the system call data while the scientific workflow was in progress. In the experiment, NPB, one of the benchmarks used for performance analysis of high-performance computing systems, was used [10]. The execution was performed using the MG workload with the highest ratio of IO-intensive tasks among various workloads of NPB. Table 1 describes the problem size and parameter values for the MG workload of NPB-OMP3.4 used in the experiment.

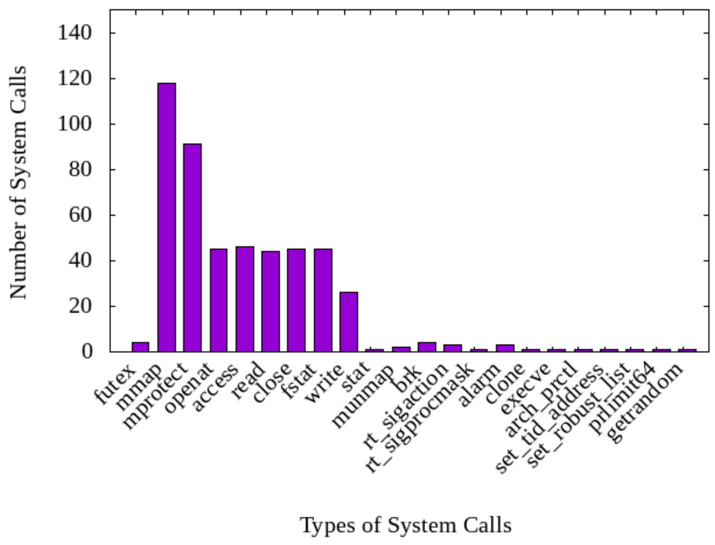

Figure 1. Number of system calls for Linux kernel build task.

Table 1. MG workload problem size and parameters.

Class	Grid Size	Number of Iterations
S	32 × 32 × 32	4
A	256 × 256 × 256	4
B	256 × 256 × 256	20
C	512 × 512 × 512	20
D	1024 × 1024 × 1024	50

As shown in Figure 2, we observed that the futex system calls increased in proportion to the size of the IO-intensive task and showed a consistent pattern. In particular, a large change in the number of system calls occurred in the section where the number of iteration operations of MG work was changed. Consequently, we confirmed that the ratio of the IO-intensive operation of the task executed in the container runtime can be derived using the number of occurrences of the futex system call.

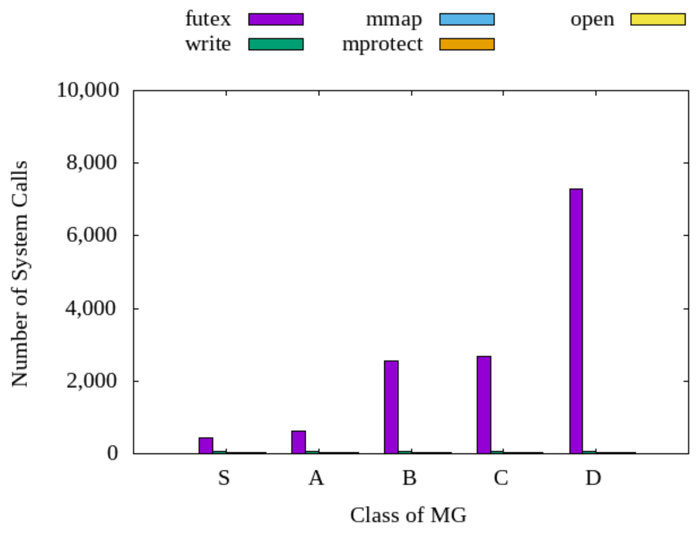

Figure 2. Number of system calls for different IO workload sizes.

2.2. *Resource Management Using Linux Cgroups*

We performed three experiments that confirm the performance of static resource management using *cgroups* in an environment where multiple containers are simultaneously operated on a single computer. In these experiments, we measured the execution time that varies according to changes in the number of containers, the size of the IO workload, and the type of workload.

Our previous study showed that the performance improvement of a container can be obtained from a framework that detects *cgroups* policy for CPU cores and automatically adjusts the number of user-level threads to the *cgroups* policy [11]. Based on our previous study, we conducted an experiment using the MG of the NPB-OMP3.4 benchmark, which dynamically adjusts the number of threads. In the experiment, the number of Singularity containers was varied, and 16 CPU cores were divided by the number of containers and evenly allocated to prepare an environment to run the MG workload. In such an environment, some workloads were executed in the situation where *cgroups* were applied, but the others were executed in the situation where *cgroups* were not applied.

As shown in Figure 3, we observed that the performance improvement rate through *cgroups* was constant even when the number of containers running the MG workload increased. As a result, we confirmed that *cgroups*-based resource management can achieve consistent performance improvement regardless of the number of containers. The next experiment was conducted to determine the performance improvement rate through *cgroups*-based resource management for various IO workload sizes in the same workload type. In the experiment, several classes with different sizes of the MG workload were constructed, and the execution time of *cgroups*-applied cases and nonapplied cases was compared.

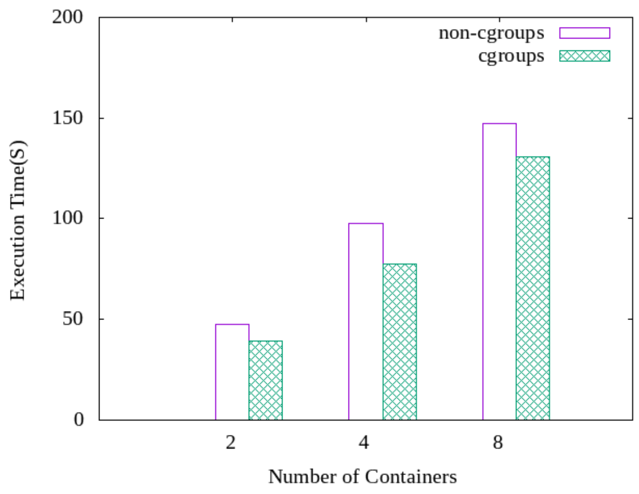

Figure 3. Performance of *cgroups*-based resource management for different numbers of containers.

Figure 4 shows that a constant performance improvement rate was confirmed regardless of the increase in the amount of MG workload. Finally, an experiment was conducted to assess the performance improvement rate of *cgroups*-based resource management according to the IO task ratio.

In general, when IO operations are performed in user-level threads, they are preempted by the kernel thread, increasing context switches and leading to performance degradation [12]. Accordingly, if containers are isolated using *cgroups*, the amount of context switches due to IO operations is expected to be reduced. In the experiment, various types of workloads of the NPB benchmark were run in the containers with or without applied *cgroups*, respectively, and the context switch occurrence information was collected. In Table 2, BT stands for a block tridiagonal solver task and IS stands for an integer sort task. Since each type of work has a different workload size, the context switch is used as a performance metric.

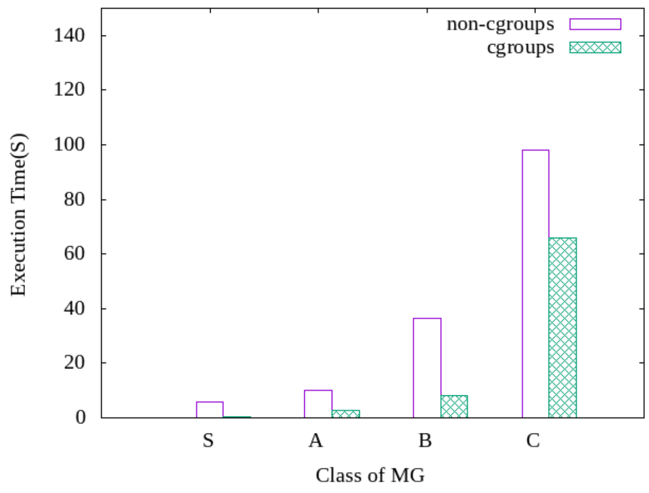

Figure 4. Performance of *cgroups*-based resource management for various IO workload sizes.

Table 2. Number of context switches for different workload types.

Task	Non-Cgroup	Cgroup	Ratio
BT	782	245	3.13
IS	919	380	2.42
MG	50,513	28,878	1.75

As shown in Table 2, the higher the proportion of IO tasks in the workload, the higher the ratio of performance improvement achieved when *cgroups* were applied. In particular, BT, which has a high proportion of IO to memory, was able to obtain significantly higher performance improvement. Through the preliminary experiments, we confirmed that the number of IO-intensive operations of the task can be determined through the number of futex system calls and that effective resource management can be performed using *cgroups*.

3. Our Solution: CSS

3.1. Design Principle

We designed CSS, a resource manager that minimizes resource contention between multiple containers, to improve system resource utilization in Singularity container runtime. The CSS was designed considering the following three principles.

1. Transparent resource management. The resource manager runs as a background service on the host operating system and has a container-independent life cycle. Users can freely use the resource manager without modifying the source code or building a new container. For this purpose, CSS collects the currently running container information based on the namespace information of the host operating system.
2. Minimized monitoring overhead. CSS is designed to minimize the overhead of collecting monitoring metrics required for container resource management. When an additional monitoring tool is introduced to measure the resource usage of a container, some overhead is added. In order to eliminate the resource management overhead and have a simple structure, CSS uses the system call log collected for the purpose of strengthening the existing security level.
3. Minimized resource coordination overhead. The cgroups technique is used for giving a policy of using computing resources to one process. When resource management is performed for all processes that are hierarchically created inside multiple containers, resource coordination overhead is caused. Therefore, to simplify the management operation, the resource management policy is copied in each container, and the copied resource management policy is applied to each container's child processes collectively. In addition, if resource management is performed very frequently, the overhead

for resource adjustment is proportionally increased. Since the appropriate resource management interval is determined according to the specifications of the computing system, an interface with which the administrator can set the resource management interval is provided.

3.2. Architecture

This section describes the structure and core components of CSS. CSS is implemented as an independent daemon in the Linux environment and operates directly on top of the operating system of the host machine; CSS has been designed based on the MAPE-K model [13]. CSS has a system call monitor, a task analyzer, and container controller modules. Figure 5 shows the structure of CSS and system components related to resource management operations in detail.

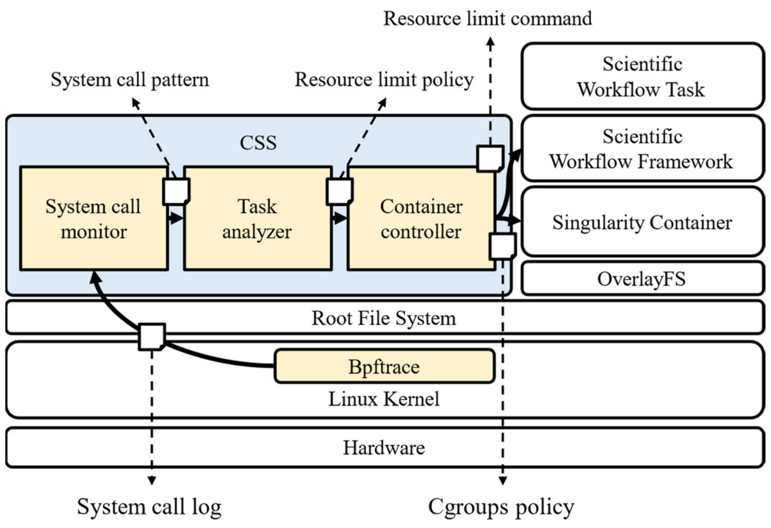

Figure 5. Architecture and resource management operation of CSS.

In the MAPE-K cycle of CSS, three modules operate sequentially. The Bpftrace [14] delivers the system call log collected from the Linux kernel to the system call monitor, and the system call monitor transforms the system call log into a system call pattern and transmits it to the task analyzer. Next, the task analyzer creates a resource limit policy by analyzing the type of resource used by the container and the system resource information together. The container controller creates a *cgroups* policy for the container based on the resource limit policy and applies the *cgroups* policy to tasks inside a Singularity container. A detailed description of the CSS core components is as follows.

1. System call monitor. It is a component with the role of collecting futex system calls to derive the resource usage type of the container. It collects the PIDs of running containers through the namespace and manages them in a tree structure. It also monitors system calls that occur in subprocesses created inside the container. The number of collected system calls is stored in the PID tree for the container. The number of occurrences of system calls is accumulated for each resource management interval, and it is initialized for each cleaning interval. The system administrator sets the resource management interval according to the physical system specification and the cleaning interval according to the size of one job.
2. Task analyzer. It analyzes monitoring data, creates resource management policies, and performs functions corresponding to the Analysis and Plan stages in the MAPE-K model. It analyzes data stored in the PID tree created by the system call monitor at each resource management interval; based on the analysis, the task analyzer creates a *cgroups* policy that determines CPU quota. Specific techniques for resource management are described in detail in the next section.

3. Container controller. It has the role of applying the resource management policy created by the task analyzer to the container. The *cgroups* policy, created by the task analyzer, starts the Singularity container runtime process as root and applies to all processes under it. The *cgroups* policy applied to the container is changed when the container type is changed, or a new container is added.

3.3. Resource Management

This section describes the detailed operation of the resource management techniques performed in the task analyzer. The proposed resource management scheme consists of SBCC and CTBRA algorithms. The SBCC algorithm classifies containers based on the number of occurrences of the futex system sampled through Bpftrace, and the CTBRA algorithm creates and applies a resource management policy based on the analysis results. Table 3 shows notations to explain the proposed technique.

Table 3. Notations.

Symbol	Description
C	$\{c_i \mid c_i$ is ith container, $1 \le i \le M\}$
R	$\{r_i \mid r_i$ is ith CPU core, $1 \le i \le N\}$
C^{io}	IO-intensive container set
C^{cpu}	CPU-intensive container set
λ	Threshold of number of system calls
μ	Number of CPU resources allocated to C^{io}-type containers
$sumSystemcall(c_i)$	A function that calculates the sum of the system calls of subprocesses created in c_i
$allocResource(c_i, start, end)$	A function that allocates CPU cores from r_{start} to r_{end} to the c_i container

3.3.1. SBCC (System Call-Based Container Classifier)

The set of all child processes of container c_i is defined as $c_i.Children$, and the inner process is defined as p_j. The number of futex system calls of p_j collected at the sampling time is defined as $p_j.NumOfFutex$. Based on this model, the $sumSystemcall(c_i)$ function is defined in Equation (1).

$$sumSystemcall(c_i) = \sum_{p_j \in \, c_i.Children} p_j.NumOfFutex \tag{1}$$

As shown in Algorithm 1, SBCC classifies task types for containers recognized by the monitoring module. C, a set of containers running on the host, and λ, the system call threshold set by the system administrator, are the input.

Algorithm 1 SBCC

```
Input: $C$, $\lambda$
Output: $C^{io}$, $C^{cpu}$
01: Collect Singularity container information and assign it to $C$
02: Collect system calls that occur inside the container
03: for $c_i$ in $C$
04:     $sum_i \leftarrow sumSystemcall(c_i)$
05:     if $sum_i \ge \lambda$ then
06:         add $c_i$ to $C^{io}$
07:         else
08:             add $c_i$ to $C^{cpu}$
09:     end if
10: end for
11: return $C^{io}$, $C^{cpu}$
```

Lines 1 and 2 of Algorithm 1 are to collect information on the currently running container and sample the system calls generated by each container. The meaning of lines

5 to 9 is to analyze the currently running container in order to classify containers into a group of IO-intensive tasks and a group of CPU-intensive tasks.

3.3.2. CTBRA (Container Type-Based Resource Allocator)

Algorithm 2 shows the CTBRA algorithm that calculates the range of resources to be allocated for each container type and performs resource allocation. It takes as input R, C^{io}, C^{cpu}, which is the result of the SBCC algorithm collected by the system call monitor, and μ, which determines the size of the minimum resource allocated to C^{io}. The result of the $allocResource(c_i, start, end)$ function is transmitted to the container controller, leading to the operation of updating the *cgroups* policy.

Algorithm 2 CTBRA

```
Input: $R$, $C^{io}$, $C^{cpu}$, $\mu$
Output: null
01: for $c_i$ in $C^{io}$
02:     if $|c^{cpu}|$ != 0 then
03:         $allocResource(c_i, 1, \mu)$
04:     else
05:         $EqualAllocation(R, c_i)$
06:     end if
07: end for
08: for $c_i$ in $C^{cpu}$
09:     $EqualAllocation(R, c_i)$
10: end for
11: return null
```

From line 2 to line 4, the algorithm allocates the minimum resource for all containers of C^{io}. If there is no CPU-intensive container, equal resource distribution is performed. However, if the CPU-intensive container and the IO-intensive container are executed at the same time, the operation of allocating the minimum resource to the IO-intensive container is performed. The algorithm between line 5 and line 9 evenly divides and allocates all CPU resources of the host machine to all containers of C^{cpu}; the equal allocation algorithm is shown in Algorithm 3. The amount of CPU resources allocated to each container is determined by the number of CPUs that the host machine has and the number of currently running containers in C^{cpu}.

Algorithm 3 Equal Allocation

```
Input: $c_i$, $M$, $N$
Output: null
01: if $N > M$ then
02:     part $\lfloor N/\ M \rfloor$
03:     $start \leftarrow i * part$
04:     $end \leftarrow start + part$
05: else
06:     $start \leftarrow i \bmod N$
07:     $end \leftarrow start$
08: end if
09:     $allocResource(c_i, start, end)$
10: return null
```

This algorithm takes as inputs c_i, the ith container; M, the number of containers; and N, the number of CPU cores in the host system. Lines 1 to 4 of Algorithm 3 indicate the situation in which the number of CPU resources is greater than the number of containers, and lines 5 to 7 indicate the situation in which countless containers are simultaneously executed. i in line 3 is the index of the c_i.

4. Evaluation

4.1. Experimental Environment

We implemented the CSS in the prototype form using the 1.16 version of the go language. The CSS operates as a daemon in the Linux OS and performs resource management by automatically recognizing Singularity containers based on namespace information. The feature of controlling resources through *cgroups* was implemented using the source code of the containerd project [15]. The system call monitoring function collects data through Bpftrace, and the collected data is processed in the monitoring module. The system used in the experiment was an IBM System x3500 M4 server. It has an Intel(R) Xeon(R) CPU E5-2650 CPU, 62 GB of memory, and 1 TB of storage. CentOS 7 with Linux 4.15 kernel was used as the operating system for the experiment. Table 4 shows the CSS parameters used in the experiment. The resource management interval and the cleaning interval are specified in seconds.

Table 4. CSS parameters used in the experiment.

Parameter	Value
λ	30
μ	2
Resource Management Interval	5
Cleaning Interval	30

Four experiments were conducted to verify the performance of the proposed CSS. The experiment was divided into two parts. The first part consisted of an experiment on NPB to verify the performance of CSS determined according to the characteristics of the task, and the second part involved an experiment on CANU to verify the effectiveness of the actually used scientific workflow task. In the NPB experiment, the workload size, the number of containers, and the heterogeneity of the concurrently performed tasks were variously configured. In addition, in the CANU experiment, the CSS parameters were set to 3 and 5, and each experiment was performed. In the experiment, the workload was run five times, and the average execution time was calculated after the execution time of each run was measured.

4.2. Performance Analysis for Various Workload Sizes

We now consider the first experiment to analyze the performance of CSS for various sizes of workloads. Among the NPB benchmark workloads, the MG workload with the highest ratio of IO operations was run, and the execution time was measured. In the experiment, the non-*cgroups* environment to which *cgroups* were not applied and *cgroups* environment in which CSS was internally executed were compared as baseline. The MG workload was subdivided into several classes based on size. Among all classes, A, B, C, and D were executed, and the execution time was compared. In order to generate some resource contention, two containers processing the MG workload were run simultaneously. Figure 6 shows the results of these experiments.

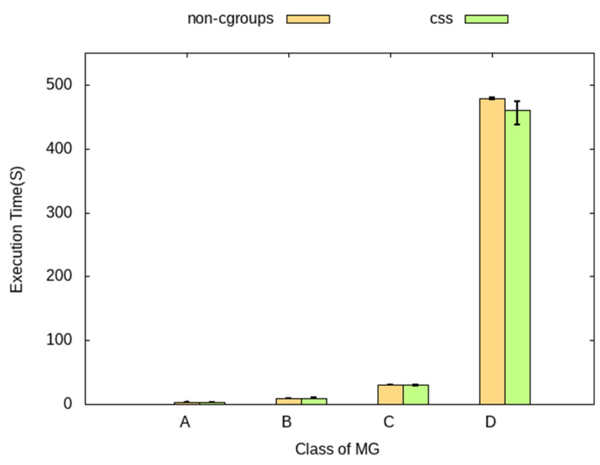

Figure 6. Performance of CSS for IO workload size.

As a result of the experiment, CSS in class A and class B took more execution time than in the non-*cgroups* environment, and class C showed a similar execution time. However, class D showed a 3% performance improvement. Through these results, we observed that CSS only achieves performance improvement in an environment where a certain level of workload size is executed.

4.3. Performance Analysis for Different Numbers of Containers

The performance was analyzed according to the resource contention level by increasing the number of containers running the workload of the same size in various ways. In the experiment, all the containers in the experiments ran the identical MG C-class workload of the NPB benchmark. The number of containers doubled from 2 to 16. Figure 7 shows the results of these experiments.

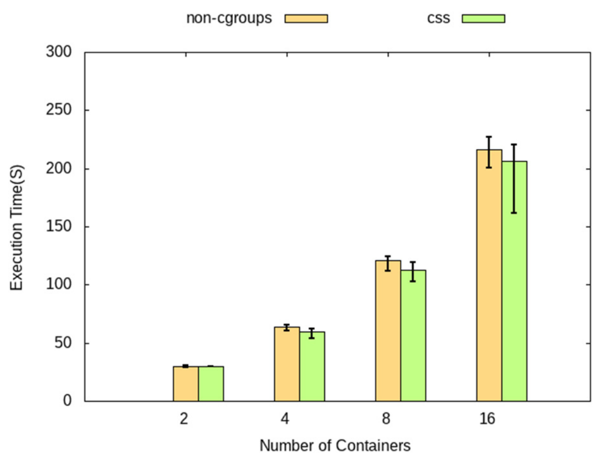

Figure 7. Performance of CSS for different numbers of containers.

As a result of the experiment, we observed that the performance increased sequentially in 2, 4, and 8 containers and decreased again in 16 containers. As a result, we confirmed that the highest performance improvement rate was observed when the number of containers and the number of CPU cores were the same. The performance improvement rate decreased in 16 containers because the CPU resources of the tasks finished relatively early are not utilized for other tasks; it causes the overall utilization rate of the system to drop.

4.4. Performance Analysis for Heterogeneity of Concurrent Tasks

In the third experiment, the performance of CSS was measured in a situation in which workloads with various types of work were executed simultaneously. Three different environments were built for comparison; the environments in which the MG workload of the NPB benchmark was run together with CG and EP, respectively, and the environment in which CG and EP were run simultaneously. Each task was executed in an independent container environment, and the workload size was C class.

The experiment was carried out in an environment where the number of concurrently executed containers was 2 or 3, and, accordingly, CSS showed lower performance than the non-*cgroups* environment. As shown in Figure 8, we observed that the types of tasks of containers running at the same time did not affect the performance of containers.

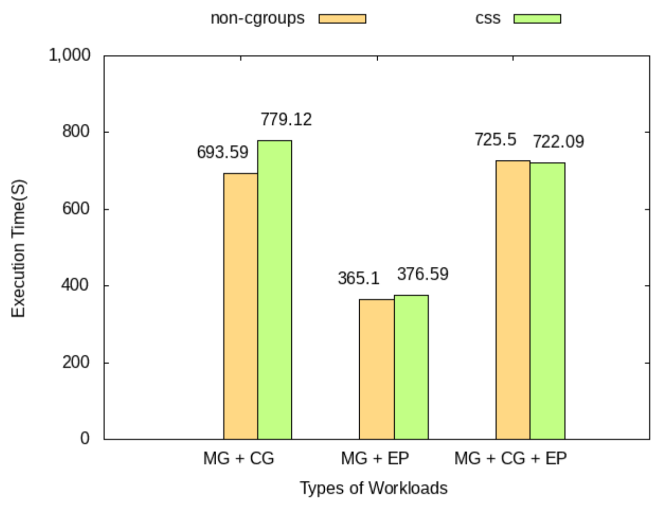

Figure 8. Performance of CSS for heterogeneity of concurrent tasks.

4.5. Validation of Scientific Workflow

An experiment was conducted to determine whether CSS was effective in the scientific workflow type application used in the actual research field. In the experiment, an open-source framework designed for the purpose of assembling high-noise single-molecule arrays called CANU was used. The experiment was conducted based on P6-C4 molecule data published by Pacific Biosciences for *Escherichia coli* K12 Interpretation [16]. In order to reproduce the container environment with a high resource contention level, the MG C-class workload was executed together in a minute unit during CANU execution. In the experiment, the execution time was measured by configuring a non-*cgroups* environment in which *cgroups* were not set as a baseline, and the resource management interval options were set to 5 or 3. Figure 9 shows the form of a scientific workflow job running on the CANU framework. In the figure, the circle means the process being created, and the label inside the circle means the number of futex system calls occurring in the process. *t* means the time in seconds since CANU was executed.

Figure 9 shows that the process structure of CANU changes flexibly at runtime and that futex system calls occur overall in processes with multiple layers. These results indicate that CANU is suitable for validating the effectiveness of CSS.

As shown in Figure 10, a performance improvement of 3.4% was obtained when the CSS interval parameter was 5, and a performance improvement of 4.5% was obtained when the interval parameter was 3. Through these experimental results, it was verified that performance improvement could be obtained without source code modification or manual resource management for scientific workflow applications used in actual scientific fields.

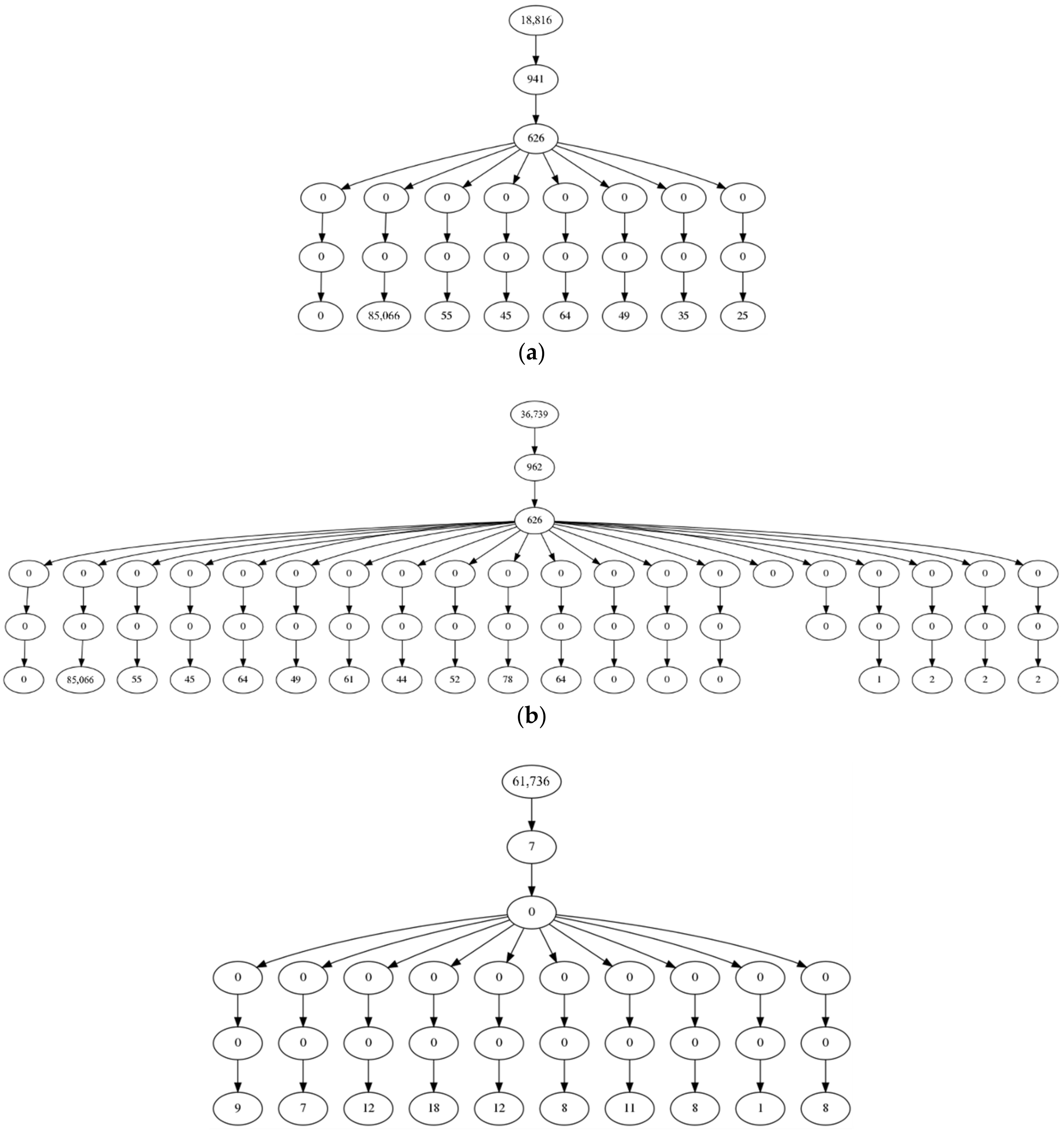

Figure 9. Forms of CANU framework work. (**a**) t = 100 (**b**) t = 200 (**c**) t = 300.

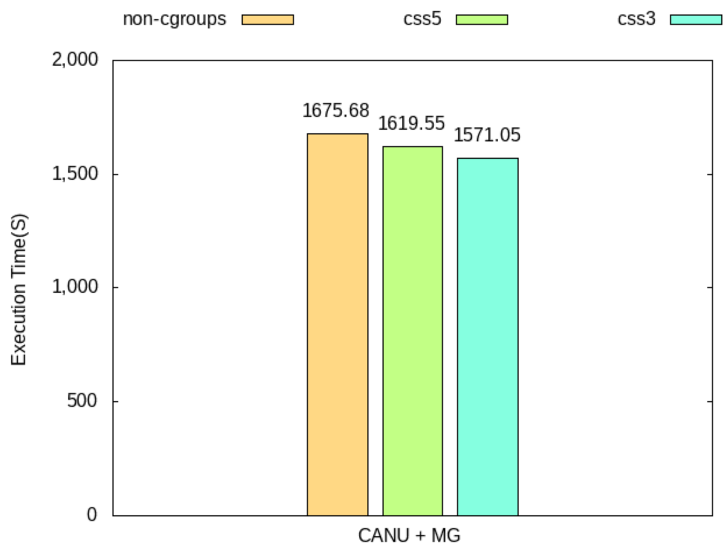

Figure 10. Performance of CCS according to interval parameters.

5. Related Works

5.1. System Call Monitoring for Container

A technology that monitors system calls made in containers has been used to track malicious attacks on the containers' runtime. Ghavamnia et al. [17], Wang et al. [18], and Kim et al. [19] proposed a technique to collect and analyze system call information that occurs in normal container operation to derive a policy and to compare the newly generated system call with the policy to block malicious attacks. When performing vertical resource management for containers in such an environment, additional monitoring overhead occurs, and the structure becomes complex.

5.2. Container Resource Management

Runsewe et al. [3] proposed a technique for performing horizontal resource management targeting data-intensive containers. This improves service availability but has a disadvantage in terms of overall system resource utilization. Al-Dhuraibi et al. [2] and Russo et al. [20] proposed a scale-up resource manager using the MAKE-K model based on the resource usage of containers. However, this resource management has a disadvantage in that additional monitoring overhead occurs. Hobson et al. [21] proposed a library that can be used to improve data communication in applications that perform scientific workflows based on containers. Such technology has a disadvantage of convenience of use because it is cumbersome to modify existing applications. Table 5 shows the result of comparing the characteristics of various container resource management techniques.

Table 5. Comparison of characteristics for different container resource management techniques.

Type of Technique	Monitoring Overhead	Complexity of System Architecture	Container Runtime
[3]	High	High	Docker
[2,20]	Middle	Middle	Docker
[21]	Middle	Middle	Singularity
CSS	Low	Low	Singularity

6. Conclusions

In this study, we proposed a dynamic resource manager for containers performing scientific workflows, called CSS, that performs simple resource management using a lightweight monitoring technique. The CSS performs MAPE-K-based resource management by applying the SBCC algorithm that classifies the resource types of containers using the system call log and the CTBRA algorithm that determines the resource usage policy for each container type in a single high-performance system. In future work, we will address the container redistribution problem by extending the system scope of CSS from one machine to a cluster. Then, in addition to the futex system call, we will find a new system call that can be used for resource management and develop an intelligent resource management technique. We will fork Singularity source code and develop new container runtime technology by implementing resource management technology utilizing seccomp technology.

Author Contributions: All the authors contributed equally to work. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korean government(MSIT) (No. NRF-2019R1A2C1006754).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Rao, V.; Singh, V.; Goutham, K.S.; Kempaiah, B.U.; Mampilli, R.J.; Kalambur, S.; Sitaram, D. Scheduling Microservice Containers on Large Core Machines Through Placement and Coalescing. In Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP), Virtual, 21 May 2021; pp. 80–100.
2. Al-Dhuraibi, Y.; Paraiso, F.; Djarallah, N.; Merle, P. Autonomic vertical elasticity of docker containers with elasticdocker. In Proceedings of the 10th International Conference on Cloud Computing (CLOUD), Honolulu, HI, USA, 25–30 June 2017; pp. 472–479.
3. Runsewe, O.; Samaan, N. CRAM: A container resource allocation mechanism for big data streaming applications. In Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Larnaca, Cyprus, 14–17 May 2019; pp. 312–320.
4. Hu, Y.; de Laat, C.D.; Zhao, Z. Multi-objective container deployment on heterogeneous clusters. In Proceedings of the International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Larnaca, Cyprus, 14–17 May 2019; pp. 592–599.
5. Tan, B.; Ma, H.; Mei, Y. A NSGA-II-based Approach for Multi-objective Micro-service Allocation in Container-based Clouds. In Proceedings of the International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 11–14 May 2020; pp. 282–289.
6. Nikkhah, S.T.; Geilen, M.; Goswami, D.; Koedam, M.; Nelson, A.; Goossens, K. A Deployment Framework for Quality-Sensitive Applications in Resource-Constrained Dynamic Environments. In Proceedings of the 24th Euromicro Conference on Digital System Design (DSD), Palermo, Italy, 1–3 September 2021; pp. 212–220.
7. Skarlatos, D.; Chen, Q.; Chen, J.; Xu, T.; Torrellas, J. Draco: Architectural and operating system support for system call security. In Proceedings of the International Symposium on Microarchitecture (MICRO), Athens, Greece, 17–21 October 2020; pp. 42–57.
8. Cgroups Web Documentation. Available online: https://www.kernel.org/doc/documentation/cgroup-v1/cgroups.txt (accessed on 22 June 2022).
9. Kurtzer, G.M.; Sochat, V.; Bauer, M.W. Singularity: Scientific containers for mobility of compute. *PLoS ONE* **2017**, *12*, e0177459. [CrossRef] [PubMed]
10. Npb Website. Available online: https://www.nas.nasa.gov/software/npb.html (accessed on 22 June 2022).
11. Song, C.; Gil, J.; Lim, J. A Performance Analysis on HPC Task Using cgroups in Singularity Container Runtime Environment. In Proceedings of the KIPS Annual Spring Conference, Seoul, Korea, 19–21 May 2022; pp. 25–27.
12. Feeley, M.J.; Chase, J.S.; Lazowska, E.D. *User-Level Threads and Interprocess Communication*; Technical Report 93-02-03; University of Washington, Department of Computer Science and Engineering: Seattle, WA, USA, 1993.
13. Kephart, J.O.; Chess, D.M. The vision of autonomic computing. *Computer* **2003**, *36*, 41–50. [CrossRef]
14. Bpftrace Github Repository. Available online: https://github.com/iovisor/bpftrace (accessed on 22 June 2022).
15. Containerd Website. Available online: https://containerd.io (accessed on 22 June 2022).
16. PacBio DevNet Website. Available online: http://pacbiodevnet.com (accessed on 22 June 2022).
17. Ghavamnia, S.; Palit, T.; Benameur, A. Confine: Automated system call policy generation for container attack surface reduction. In Proceedings of the 23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID), San Sebastian, Spain, 14–15 October 2020; pp. 443–458.
18. Wang, X.; Shen, Q.; Luo, W.; Wu, P. RSDS: Getting System Call Whitelist for Container Through Dynamic and Static Analysis. In Proceedings of the 13th IEEE International Conference on Cloud Computing (CLOUD), Virtual Event, 18–24 October 2020; pp. 600–608.

19. Kim, S.; Kim, B.; Lee, D. Prof-gen: Practical Study on System Call Whitelist Generation for Container Attack Surface Reduction. In Proceedings of the 14th IEEE International Conference on Cloud Computing, (CLOUD), Chicago, IL, USA, 5–10 September 2021; pp. 278–287.
20. Russo, G.R.; Cardellini, V.; Casale, G.; Presti, F.L. MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing. In Proceedings of the International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 10–13 May 2021; pp. 314–323.
21. Hobson, T.; Yildiz, O.; Nicolae, B.; Huang, J.; Peterka, T. Shared-Memory Communication for Containerized Workflows. In Proceedings of the International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, Australia, 10–13 May 2021; pp. 123–132.