2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)

Should PARSEC Benchmarks be More Parametric? A Case Study with Dedup

Carlos A. F. Maron*†, Adriano Vogel*, Dalvan Griebler*†, Luiz Gustavo Fernandes*
* School of Technology, Pontifical Catholic University of Rio Grande do Sul (PUCRS), Porto Alegre, Brazil
†Laboratory of Advanced Research on Cloud Computing (LARCC), Três de Maio Faculty (SETREM), Três de Maio, Brazil
Email: {carlos.maron, adriano.vogel, dalvan.griebler}@acad.pucrs.br, luiz.fernandes@pucrs.br

***Abstract*—Parallel applications of the same domain can present similar patterns of behavior and characteristics. Characterizing common application behaviors can help for understanding performance aspects in the real-world scenario. One way to better understand and evaluate applications' characteristics is by using customizable/parametric benchmarks that enable users to represent important characteristics at run-time. We observed that parameterization techniques should be better exploited in the available benchmarks, especially on stream processing domain. For instance, although widely used, the stream processing benchmarks available in PARSEC do not support the simulation and evaluation of relevant and modern characteristics. Therefore, our goal is to identify the stream parallelism characteristics present in PARSEC. We also implemented a ready to use parameterization support and evaluated the application behaviors considering relevant performance metrics for stream parallelism (service time, throughput, latency). We choose Dedup to be our case study. The experimental results have shown performance improvements in our parameterization support for Dedup. Moreover, this support increased the customization space for benchmark users, which is simple to use. In the future, our solution can be potentially explored on different parallel architectures and parallel programming frameworks.**

I. Introduction

Benchmarks are relevant tools for several computing areas. Either in the industry or in the academia, benchmarks are synthetic programs with real-world characteristics used for evaluating and comparing the performance of hardware and computing systems [1]. Hence, benchmarks contribute for improving the existing technologies as well as for developing new hardware and software systems. Many benchmarks are available with different characteristics and purposes, divided in two main categories: benchmarks and microbenchmarks. A microbenchmark is designed for evaluating a specific part of a system [2]. On the other hand, a benchmark (a.k.a. application benchmark, synthetic benchmark) is more complex and elaborated, composed by actual application traces combined with functions demanding different system performance.

The performance of a benchmark can be affected by several aspects [2]. The performance of benchmarks is mostly affected by the the processing architecture, which can be slower or faster. Also, the performance can vary according to the used technologies and execution environments. In order to limit the scope, some benchmarks are used only for evaluating the performance capabilities of the architectures. However, the code functions and variables of benchmarks also have performance variations because the codes can be customized simulating different behavior during execution. These benchmark behaviors simulate real application's characteristics resulting in performance variations, caused by: different instructions, code variables, several data types and sizes, and different processing algorithms. The application characteristics are also relevant to be simulated and evaluated. However, deal with several characteristics relevant for performance may be a difficult and time consuming task for benchmark users. One way for reducing the complexities involved, is enabling parameterization support in benchmarks [2].

Stream processing application [3], [4], [5], [6] became a significant workload in our computing systems, represented by the processing of images, audio, video, signals, etc. The stream processing applications have unique processing behaviors that is a result of the combination of different aspects, such as: sliding windows, communication channels, buffers, continuous flow of data [5]. Moreover, Hirzel *et al.* [5] highlighted the need for research and optimization of applications and benchmarks for the stream processing context.

Considering the related literature, Eigenbench [7] is a microbenchmark for evaluating transactional memory systems. In Eigenbench, the goal is to simulate application behaviors through parameters defined by the user before the execution of the microbenchmark, enabling users to simulate realistic application behaviors. Although Eigenbench is a parametric microbenchmark, it does not represent the domain of stream parallelism and consequently, there is no support to evaluate realistic behaviors for stream processing. Moreover, StreamIt [8] is a programming language and compiler targeting stream processing applications, the language provides programming abstractions and the compiler applies performance optimizations. Although StreamIt environment (benchmarks and architecture) provides parametric configurations, such benchmarks are not fully portable by only supporting the StreamIt language and architecture. Moreover, StreamIt benchmarks represent only the dataflow and data stream scenario. Differently, in this work we focus on stream parallelism abstractions on representative benchmarks from PARSEC, providing additional stream processing metrics, parameters, and input sets.

The PARSEC benchmark suite [9] is the state-of-the-art for evaluating multi-core computing architectures. On one hand, stream processing application are characterized with fluctuations regarding: execution behavior, environment, input rates,

2377-5750/19/$31.00 ©2019 IEEE
DOI 10.1109/PDP.2019.00038

and the types of data processed. On the other hand, PARSEC executions are characterized with a static and regular workload trend, not able to fully represent the characteristics of stream processing applications. In this work, we aim at implementing and evaluate the feasibility of turning stream processing more realistic through the implementation of parametric benchmarks not addressed by previous efforts. The main contributions of this work are the following:

- **Evaluation of stream parallelism characteristics present in the PARSEC suite.** Although there are studies available [10], [11], [12] showing that PARSEC benchmarks can be optimized, such works do not address stream processing characteristics. This study goes beyond parallel programming frameworks and parallelism strategies. We identified characteristics that affect the applications' behavior and we demonstrated how such characteristics may be tuned.
- **Support to new parameters in the PARSEC suite from stream processing applications.** Dedup is a benchmark of the PARSEC suite that have characteristics relevant for the stream parallelism context. Consequently, these benchmarks were customized and extended for supporting a parameterizable execution. The original version only supported parameters regarding the number of threads and input size. Our implementation supports the customization on the buffer sizes, queues size, size of stream elements and sliding window size, as well as more data formats and input sizes are now supported.
- **A performance analysis of how the parameterization affects the execution of PARSEC benchmarks** The evaluation compares the proposed implementation to the default benchmark implementations. We implemented a monitoring mechanism that traces the benchmark execution considering relevant metrics for stream parallelism, such as throughput, latency, and service time.

This paper is organized as follows. Section II shows the implemented strategies. The evaluation methodology as well as the results are presented in Section III. Finally, Section IV emphasizes this study's conclusions.

II. Design and Implementation

Stream parallelism is a concept related to the paradigm of stream processing [5], [8]. Dedup was chosen in this work for exploiting and extending stream parallelism characteristics. Dedup was modified in their original POSIX threads implementations in order to enable new parametric executions.

Figure 1 shows relevant stream processing characteristics that can be parametric. Considering the stream processing scenario and the benchmark used in this work, the sliding window represents the number of elements handled in stages queues. The operations are performed when a given computation has to pop or push elements in the queues. Usually, buffered stages use a buffer size close to the sliding windows size, Dedup implements buffers for communicating between stages.

Dedup is a benchmark of the data compression and deduplication domain, which combines techniques of local and global

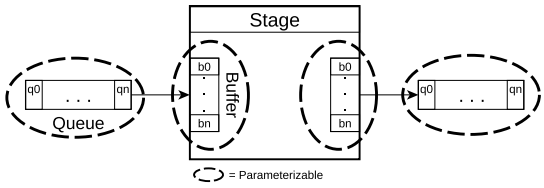

Fig. 1. The highlighted regions represent characteristics parameterizable on stream parallel applications.

compression. The original POSIX Threads version implements a pipeline with five stages. The first stage performs the files reading with its their fragmentation. The second stage performs a refined data fragmentation and create chunks that are sent to next stage. The third stage (*DD* in Figure 2) generates hash identification for each chunk. The identification are stored in a global table which are compared to the other hashes. When a duplicate hash is found, the chunk related to this hash is sent to the last stage. The fourth stage receives the chunks that are not duplicated and performs a local compression of them. This compression is sent to the last stage. The last stage is in charge of reordering the out-of-order chunks as well as writing an output file. The input set provided by PARSEC is formed by 6 TAR files containing only DAT and ISO file formats. The TAR files vary in size from 10 KB to 705 MB. Moreover, the default implementation supports only one input file per execution.

Figure 2 represents the new Dedup version to support the parameterization. This was necessary to use files larger than 1GB as well as for reading directories with several files instead of reading a single file. The new version reads in the first stage from the directory provided as argument, represented by the stage `input` in Figure 2.

The characteristic of only processing TAR files from the original version was maintained. However, in order to process more files concurrently, the first fragmentation performed in the INPUT stage was removed for avoiding the creation of an internal state (stateful) in the next stages. The fragmentation in the first stage was performed to reduce the cost of disk reading by creating static chunks of 128MB [9]. Additionally, the search for deduplication performed in the DD stage was optimized with a dedicated HASH table for each file. Each file has a HASH table to avoid the counter-productive (*e.g.,* very large data bases) comparison of chunks from different files.

The evaluation of the new Dedup implementation demanded additional input sets. We created 4 new input sets organized in classes named *Test Class, Light Duty Class, Heavy Duty Class and Free Class*. The input classes *Light Duty* and *Heavy Duty* are divided in 4 subclasses according to the file formats supported: L1 (Light) and H1 (Heavy) and 1 referring to images, L2 and H2 text files, L3 and H3 audio and video files, and LS and HS (shuffled) composed by a mix of file formats and sizes. Moreover, the *Light Duty* is meant for computing architectures with limited resources (*e.g.,* PCs or small servers). *Heavy Duty* is supposed to be used on high-

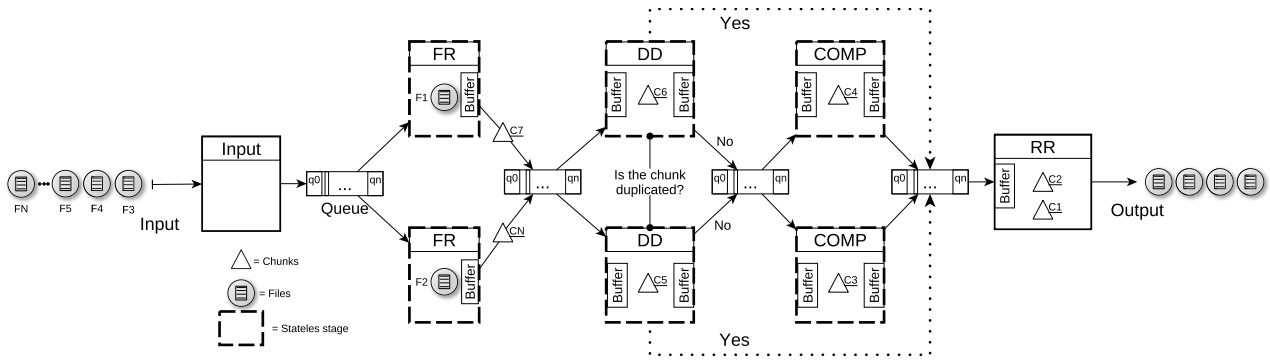

Fig. 2. **Overview of Modified Dedup**. First Stage Input. Second Stage Fragment Refine (FR). Third Stage *Deduplication* (DD). Fourth Stage *Compression* (COMP). Fift Stage *Reorder* (RR). Arrows: ordered stream. Dashed stages are parallel. The illustration represents Dedup with a parallelism degree of 2.

performance architectures. The *Test Class* is suitable for using when the goal is to only test if the input works, while the *Free Class* is customizable by the user to specify the file format as well as the number of files. On each class, the user may define the number of input files according to the number of files inside the read directory.

The default size of the queues in Dedup is fixed with over 1 million positions. The number of queues between the stages scales proportionally to the number of replicas used. The queues size was maintained the default configuration, because this aspect is not considered as relevant as the buffer sizes and the sliding window for the stream processing scenario. Consequently, we supported the customization regarding the buffer sizes, which also sets the sliding window size.

Another relevant aspect in Dedup is its chunk size. In data compression, the chunk represents the data granularity used. In stream processing, a chunk is handled as one stream element. For our stream processing scenario, we enabled a parametric chunk size to the end users. The chunk size affects the size and amount of data processed, which is defined in the BSW fragmentation algorithm [13] inside the second stage. Table I describes the new arguments supported by Dedup execution and usage examples.

TABLE I
OVERVIEW OF PARAMETERS SUPPORTED ON DEDUP.

Argument	Description and Options	
parsec_stream	Manager script. Options: -i, -r, -p, -a	
-i	Input Classes. Options: free,test,h1,h2,h3,hs,l1,l2,l3,ls	
-r	Number of replicas	
-p	Program. Ex: -p dedup	
-a	Actions for programs parameterization:	
	Dedup	
	fr	Buffer and sliding window (SW) output of fragmentation stage
	dd	Buffer and SW Deduplication stage
	comp	Buffer and SW Compression stage
	rr	Buffer and SW Reorder Stage
	chunk	Window Size for fragmentation algorithm (Element size)
	trace	
	notrace	
Examples: $ parsec_stream -p dedup -r 12 -i h1 -a chunk 2048 fr 40 dd 40 comp 40 rr 40 notrace $ parsec_stream -p dedup -r 12 -i h1 -a chunk 2048 fr 40 dd 40 comp 40 rr 40 trace 1		

III. RESULTS

In this section we present results obtained using the parameterized implementation on Dedup comparing with the original version, used as a baseline. All code versions use the C language and parallelism is exploited using the POSIX Threads library. The benchmarks were modified for enabling parametric requirements of stream processing characteristics as well as for supporting different workload trends at runtime. Moreover, customization were needed for implementing performance metrics that are relevant for this application domain, such as throughput, latency, and service time [14].

A. Methodology

Dedup was evaluated considering throughput, latency, and service time. The performance was measured and presented as an average of 10 runs along with its standard deviation.

The Heavy Duty class was used in the benchmark execution with its subclasses described in Section II. The Dedup subclasses were customized with files duplicated in order to increase the input set and maintain a similar size in all subclasses. In Table II are showed all subclasses used in the test and the stream characteristics that are parameterizable in Dedup. The proportion of subclasses was chosen aiming to approximate the total size of the sets on each subclass. For instance, to achieve a size of 6GB in the HS we needed to add more files, because our goal was to vary input rate and type.

The tests were run in a machine with two processors Intel Xeon E5-2620 v3 working at 2.40 GHz (12 cores and 24 threads), 32 GB of RAM. Moreover, Ubuntu Server 64 bits (kernel 4.4.0-121-generic) was the operating system used. The benchmarks were compiled with GCC 5.4.0 using the compiler flag -Os[1].

The baseline used the values of default PARSEC version, which are: sliding window (SW) with value 20; buffer size of 20 indices, and window size (WS) 32. PARSEC documentation lacks information and reasons for using such values. The Dedup version with parametric support was run with the following characteristics: sliding window (SW) with value 40; buffers with maximum of 40 indices and the window size (WS) used was 2048 (fragmentation algorithm). The buffer sizes are

[1]Adds size optimizations, -Os also includes all -O2 optimizations.

TABLE II
PARAMETRIC SCENARIO FOR DEDUP.

Parameterizable Stream Characteristics	sliding window, queue, buffer, stream element size (chunk fragmentation), stream data type (text, image, video, audio)			
Input	H1	H2	H3	HS
Number of Files	15	15	15	34
Total size (GB)	6.2	6.5	6.3	6.0

related to the input and output on the stages. In the FR stage, only the output buffer is adjusted because the input buffer is used for work scheduling. On the other hand, in the reorder (RR) stage only the input buffer was adapted, which occurs in the last stage.

The Figure 3 shows the throughput and service time on Dedup parameterized with the input set H1. This result shows that the parameterized version enabled Dedup to improve its throughput rate as well as reducing its service time. Dedup is a benchmark were throughput is the most relevant performance metric [15]. Consequently, the latency results are not presented due to space limitations.

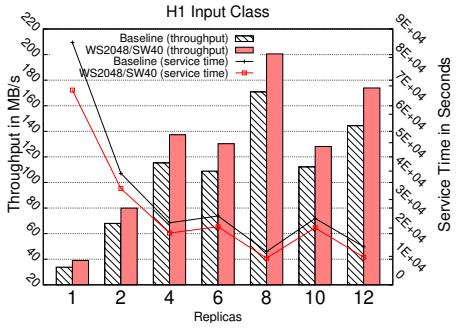

Fig. 3. Dedup performance using the H1 input.

Dedup tends to achieve a scalable performance with different input sets. The use of hyper-threading on executions with 6 replicas shown to reduce the performance in both scenarios. In Dedup, the parallelism degree is defined as an argument and represents a value smaller than the number of threads allocated by the system. This calculation considers: $3 + 3n$, where n is the minimum number of threads defined by the user for a given stateless stage.

Applications using pipeline parallelism naturally demand intensive communication between active threads. The use of 8 replicas was the degree of parallelism that achieved the highest throughput due to the balance between computations and communication. Using 8 threads, the bottleneck of the slower stages was reduced by replicating them, avoiding performance degradation caused by too many threads competing for resources. The parameterized version of Dedup improved the performance by reducing the number of chunks (larger sizes) and consequently, the amount of communication. More computations were done at the price of less communication. Moreover, larger buffer sizes were tested, which also reduced the communication among stages and avoided recurrent accesses to the communication channels that are shared by the replicas. Such shared communication channels are managed by locks. The buffers may be viewed as dedicated queues where each replicas can access it without needing locks.

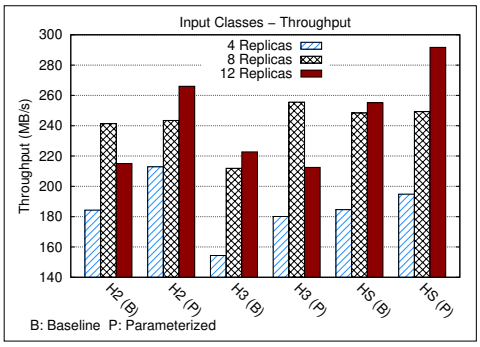

Fig. 4. Dedup Throughput on Different Input Classes.

In Figure 4 is shown the results from Dedup with the other input set. Despite the contrasts between the inputs provided, the performance trends are similar. The parameterized version outperformed the baseline default PARSEC version. The one exception occurred in the H3 with 12 replicas, where the baseline was slightly better than the new version.

Table III shows the Dedup results considering the different scenarios and input sets. Despite the fact that the parameterized implementation used larger chunks that in some cases can reduce the precision, the parameterized version achieved a better precision in 50% of cases. Additionally, the parametric version was more effective by reducing the data size after compression. Moreover, another aspect showing the parameterized implementation is performing well is the fact that it created less chunks, which improved its performance and efficiency.

In our experimental results was shown that the implementation provided to users a starting point for indicating which relevant stream parallelism characteristics can affect the behavior and performance of their applications. The implementations and the results presented are meant for users concerned with their application behavior and performance. Consequently, several behaviors of the application in a given environment can be evaluated and predicted using benchmarks. A relevant example of using the implementations provided is on resource slicing in a dynamic and flexible environments (*e.g.,* cloud environment) for stream processing applications. In case of a user with no performance expertise, a parametric benchmark is a tool that provides meaningful insights of performance in the context of stream parallelism, where realistic characteristics are defined as benchmark parameters.

IV. CONCLUSION

In this work, we supported a stream processing benchmark with customizable configurations, enabling application programmers to exploit representative executions. This was

TABLE III
DEDUP RESULTS OVERVIEW.

Input Set	Chunks		Chunks Duplicated		Chunks Medium Size (KB)/ STDev (KB)		Output Size After Deduplication (GB)		Output Size After Compression (GB)	
	B	P	**B**	P	**B**	P	**B**	P	**B**	P
H1	1653279	1078412	5.86%	1.17%	3.91 / 5.11	6.00 / 4.13	5.98	6.10	5.86	5.97
H2	1436500	1156488	22.40%	35.52%	4.70 / 4.95	5.84 / 3.76	4.29	4.28	3.69	3.65
H3	1645041	1092186	31.54%	28.23%	3.98 / 5.13	6.00 / 4.16	4.34	4.48	4.31	4.44
HS	1477071	1055706	26.39%	27.89%	4.23 / 4.93	5.92 / 3.87	4.19	4.37	3.98	4.15
B: Baseline. P: Parameterized										

performed by implementing parametric funcionalities and performance metrics that improved the usability and can be easily used by application programmers. Relevant characteristics of stream parallelism were implemented as parameters for supporting customization to users. We implemented the parametric capabilities to the benchmark Dedup from the PARSEC suite. Moreover, new input sets and classes were provided in order to simulate a more representative workload for stream processing applications. These inputs were tested in different implementations and compared to default PARSEC version. Using this new test scenario, the benchmark provided a better representation of stream processing applications. Based on our implementations and results achieved, we conclude that when considering the scenario of stream parallelism, the PARSEC benchmarks should be more parametric to support different and custom application characteristics.

In the original version, the PARSEC goal is to evaluate the multi-core architectures, while we consider as a relevant aspect the application characterization and benchmark in a given execution environment. The implementation demonstrated that it is possible to customize existing benchmarks toward a more recent and representative application's characteristics. We have shown that it is possible to collect performance metrics at run-time, such as throughput, latency, and service time. The metrics can be collected and visualized during the execution by only setting a *trace* flag. However, all these metrics may be only relevant for the stream processing scenario. We contributed with performance aspects of stream parallel applications as well as with the work of the PARSEC suite. We aim at extending this work in terms of additional parametric functionalities for stream parallelism in other benchmarks from the PARSEC suite. Additional performance tuning can be achieved in the PARSEC suite by improving data locality and by optimizing the degree of parallelism on replicated stages.

ACKNOWLEDGMENT

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nivel Superior - Brasil (CAPES) - Finance Code 001, FAPERGS 01/2017-ARD project PARAELASTIC (No. 17/2551-0000871-5), and PUCRS School of Technology. We would like to thank Laboratório de Alto Desempenho (LAD) from PUCRS for providing computing resources.

REFERENCES

[1] J. A. Arnold, K. Huppler, K.-D. Lange, J. L. Henning, P. Cao *et al.*, "How to Build a Benchmark," in *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering*. ACM, 2015, pp. 333–336.

[2] A. Blumenthal, M. Luedde, T. Manzke, B. Mielenhausen, and C. E. Swanepoel, "Measuring Software System Performance Using Benchmarks," Jun. 9 2009, uS Patent 7,546,598.

[3] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "StreamIt: A Language for Streaming Applications," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 179–196.

[4] H. C. M. Andrade, B. Gedik, and D. S. Turaga, *Fundamentals of Stream Processing: Application Design, Systems, and Analytics*, 1st ed. New York, NY, USA: Cambridge University Press, 2014.

[5] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm, "A Catalog of Stream Processing Optimizations," *ACM Comput. Surv.*, vol. 46, no. 4, pp. 46:1–46:34, Mar. 2014.

[6] D. Griebler, R. B. Hoffmann, M. Danelutto, and L. G. Fernandes, "High-Level and Productive Stream Parallelism for Dedup, Ferret, and Bzip2," *International Journal of Parallel Programming*, pp. 1–19, February 2018.

[7] S. Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun, "Eigenbench: A Simple Exploration Tool for Orthogonal TM Characteristics," in *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, Dec 2010, pp. 1–11.

[8] S. Thies, William; Amarasinghe, "An Empirical Characterization of Stream Programs and Its Implications for Language and Compiler Design," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 365–376.

[9] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[10] A. Navarro, R. Asenjo, S. Tabik, and C. Cascaval, "Analytical Modeling of Pipeline Parallelism," in *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 281–290.

[11] D. Chasapis, M. Casas, M. Moretó, R. Vidal, E. Ayguadé, J. Labarta, and M. Valero, "PARSECSs: Evaluating the Impact of Task Parallelism in the PARSEC Benchmark Suite," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 41:1–41:22, Dec. 2015.

[12] D. De Sensi, T. De Matteis, M. Torquati, G. Mencagli, and M. Danelutto, "Bringing Parallel Patterns Out of the Corner: The P3 ARSEC Benchmark Suite," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 33:1–33:26, Oct. 2017.

[13] A. Muthitacharoen, B. Chen, and D. Mazières, "A Low-bandwidth Network File System," in *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '01. New York, NY, USA: ACM, 2001, pp. 174–187.

[14] A. Vogel, D. Griebler, D. D. Sensi, M. Danelutto, and L. G. Fernandes, "Autonomic and Latency-Aware Degree of Parallelism Management in SPar," in *Euro-Par 2018: Parallel Processing Workshops*. Turin, Italy: Springer, August 2018, p. 12.

[15] X. Zhan, Y. Bao, C. Bienia, and K. Li, "PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X," *SIGARCH Comput. Archit. News*, vol. 44, no. 5, pp. 1–16, Feb. 2017.