Available online at sjuoz.uoz.edu.krd

journals.uoz.edu.krd

Science Journal of University of Zakho

Vol. 6, No. 3, pp. 118 –123, Sept.-2018

p-ISSN: 2410-7549
e-ISSN: 2414-6943

A STATE OF ART SURVEY FOR OS PERFORMANCE IMPROVEMENT

Lailan M. Haji [a, *], Subhi R.M. Zeebaree [b], Karwan Jacksi [a], Diyar Q. Zeebaree [c]

[a] Faculty of Science, University of Zakho, Zakho, Kurdistan Region, Iraq - (Lailan.haji, karwan.jacksi)@uoz.edu.krd
[b] Technical Informatics College-Akre, Duhok Polytechnic University, Kurdistan Region, Iraq – (subhi.rafeeq@dpu.edu.krd)
[c] School of Computing, Faculty of Engineering, University Teknologi Malaysia (UTM), Johor, Malaysia

Received: Jul. 2018 / Accepted: Sept., 2018 / Published: Sept., 2018 *https://doi.org/10.25271/sjuoz.2018.6.3.516*

ABSTRACT:
Through the huge growth of heavy computing applications which require a high level of performance, it is observed that the interest of monitoring operating system performance has also demanded to be grown widely. In the past several years since OS performance has become a critical issue, many research studies have been produced to investigate and evaluate the stability status of OSs performance. This paper presents a survey of the most important and state of the art approaches and models to be used for performance measurement and evaluation. Furthermore, the research marks the capabilities of the performance-improvement of different operating systems using multiple metrics. The selection of metrics which will be used for monitoring the performance depends on monitoring goals and performance requirements. Many previous works related to this subject have been addressed, explained in details, and compared to highlight the top important features that will very beneficial to be depended for the best approach selection.

KEYWORDS: OS performance, thread, multiprocessor, transaction memory.

1. INTRODUCTION

No matter how fast our hardware gets, the performance of our system always matters. Personal Computers (PCs) were introduced that gave users the ability to perform their programs on demand without having to wait for the mainframe scheduler to execute their programs. Accordingly, PCs were slowly reducing the prominence of the mainframes while they are dramatically getting more powerful (Zeebaree & Jacksi, 2015). Programmers tend to add complexity and sophistication to their systems for matching any hardware performance improvements. So, there is always a need to design software that achieves good performance on whatever hardware is available to us. Perhaps the first and most important step in determining the performance of any system is achieving clarity about what anybody really cares about.

In a Massively Parallel Processing system, which contains a large number of processors, there is increased competition for accessing the common resources (Rashid, Sharif, & Zeebaree, 2018). The factor that is chosen to characterize the performance of systems is called metrics. Obviously, their proper choice is of critical importance. Just because some quantity is measurable, however, does not make it a good metric. Environment state in which applications run predominantly needs to be monitored. For example, the online availability of a server to receive clients' requests might be of importance to an application to know. By monitoring threads, a framework is provided to monitor status changes in parts of a system which cannot be monitored by listening for events. Performance monitoring cycle consists of four stages which are: monitoring, planning, developing and rating, as shown in **Figure 1**.

A monitor definition is a thread-safe object, module, or class which wraps around a mutex so that more than one thread can safely access a variable or a method. The defining characteristic of a monitor is that its methods are executed with mutual exclusion: in any point of the time, at max one thread has the ability to execute any of its methods. The monitor can give the threads the ability to wait for a particular situation by using one or more condition variables ("Monitor (synchronization)," 2018).

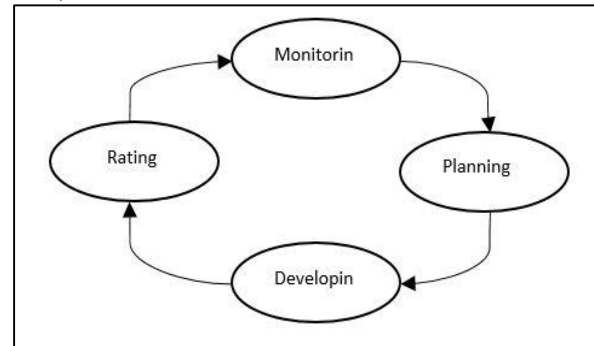

Figure 1. Performance monitoring cycle

For justifying performance monitoring, there are different reasons that execution tuning of a subsystem or an application is as yet justified regardless of the exertion. Even in a period where equipment is viewed as "shabby" and labor is viewed as "costly". The principal case is when PCs are obtained for a huge number of coins so that even economies of only a couple of percent can adjust for the pay rates of the general population doing execution work. The other case is when PCs reaches the thermal limit and maximum power to the point where no more servers can be installed. But when farther capacity is wanted, the user may either be forced to change the hardware with more sophisticated ones (if it exists), or the performance has to be modified to get as much performance as possible. The additional motivation is, without a doubt, the personal pride of the software designer/programmer. Typically, performance analysis is needed to be performed during the entire development cycle of an application, so that application does not exhibit "inefficient" behavior or excessive consumption of computing resources (Jarp, Jurga, & Nowak, 2008).

* Corresponding author
This is an open access under a CC BY-NC-SA 4.0 license (https://creativecommons.org/licenses/by-nc-sa/4.0/)

The main challenge is that trendy languages and systems which offer very low support for guaranteeing the properties of concurrency correctness, multithreaded determinism, and sequential consistency since the majority of the existing approaches are not serviceable. Software-based and dynamic methods slow programs by up to an order of magnitude since synchronization is required for capturing and controlling the dependences of cross-thread at nearly every access to probably shared memory. A crucial challenge to programs of systems for thread monitoring is "how to partition tasks of applications and mapping them to one of many possible core-thread configurations" to get the wanted performance in terms of delay, power, resource consumption, and throughput.

When increasing the number of threads and cores, as a result, the number of mapping choices increases exponentially. When threads migrate, the relating information more often remains in the first memory module and the migrated thread access it remotely. Reallocating threads to a place near to where their data are stored can help to mitigate those problems. The objective of this paper is to show a modern and up-to-date view of OS performance. This paper is organized as follows; Section 2 reviews each of the systems for performance-enhancing; Section 3 provides a comprehensive review of the systems and finally, we draw some conclusion in Section 4.

2. LITERATURE REVIEW

Performance metrics of the Operating system are linked to the performance of processor, memory, network, and disk. Performance requirements and monitoring goals are the criteria for choosing the metrics that will be used for monitoring the performance. Some of the OS performance evaluation metrics are discussed.

2.1 Thread Mapping

Thread mapping is a widely known technique for memory affinity. It places threads to specific cores to reduce memory latency or alleviate memory contention. (Ju, Jung, & Che, 2015), offered a methodology for thread-level modeling to meet the challenge of a way to value the performance of assorted attainable mapping of program-task-to-core selections throughout the initial programming section, once the feasible program is nevertheless to be developed. The main idea was to ignore the micro-architectural and instruction-level details and only model thread-level activities, excluding those having a major effect on the performance at thread-level. Moreover, the modeling of thread-level is way coarser than instruction-level modeling. By these features, the methodology is more responsive for fast evaluation of the performance of a huge number of mapping choices of the program-task-to-core within the phase of initial programming.

Depending on a huge number of samples of code; case studies are accessible in workbenches of IXP1200/2400; simulation results showed that the maximal ongoing line rates predestined using the simulation tool is 6% and it gets up to 8% by using the queuing network of cycle-accurate. There is a component which has not been tended to in their explanatory displaying method, i.e., dynamic multithreading, the threads quantity may shift at various stages of program execution.

(Petrucci et al., 2015a) offered an optimization approach that consisted of an optimization model of Integer Linear Programming (ILP), and a scheme to set the thread-to-core dynamically. Analysis of simulation present performance gains and energy savings for an assortment of workloads, as compared with schemes of state-of-the-art. Benefiting from the capabilities of Linux scheduling and performance-monitoring, implementation and evaluation of a prototype of the approach were done in thread assignment at the user level. According to results of the simulation, the approach accomplishes huge performance gains and energy savings for an assortment of workloads; as well as; performs much better than other proposed schemes of thread assignment that do not address memory transfer speed requirements. For example, depending on the workload the improvement of Energy Delay Product (EDP) over the state-of-the-art is about 10% to 40%. Furthermore, experimental evaluation and implementation of the scheme was done on a real system of heterogeneous multicore. The approach effectively fulfilled thread performance and memory bandwidth requirements for a diversity of workloads consisting of programs of SPEC benchmark. The result showed that EDP gains the average of 15% to 35% maximum for an assortment of workloads. The performance may defer significantly between different configurations and mixtures of block and grid sizes, although they may give the highest occupancy.

To select the most beneficially block size automatically, (Connors & Qasem, 2017) proposed the construction of an ML-based heuristic. For driving the tasks of feature selection, training data generation, feature extraction, evaluation, model training, and selection, the framework generates custom scripts. Executing the scripts comes after generating the custom Makefiles, then a ProgList file is created by the Configurer which contain the needed information to create training data on the target platform. Cross-fold validation is used to evaluate the performance of the support vector machine (SVM) model which by using the training set is trained. Using events of dynamic performance as features and supervised Machine Learning (ML) algorithms, the model foretells if the performance will have an improvement for a given kernel if any change in block size occurred. The researchers had the problem of not having sufficient programs to build a training dataset, that is sufficiently large and diverse which is a common issue. As a conclusion of the research, they found that different block sizes are needed due to subtle differences in a kernel's runtime behavior. Furthermore, choosing the block size of thread is delicate to memory access patterns.

(Pandey & Sahu, 2018), have proposed an effective mapping for virtual page to memory slice, used simulated annealing-based thread to core mapping of multi-threaded application onto 3D stacked memory chip-multiprocessor. To minimize the cost of communication of the on-chip core to core, the researchers mapped the multithreaded application's N threads to chip-multiprocessor's N cores in the thread to the core mapping process. Meanwhile, in memory mapping, they minimized the cost of on-chip communication among all the cores, of virtual page access. The virtual page slice mapping to DRAM; comparing to the thread mapping to core; gets lower priority. Sniper simulator (version 6.1) was used to evaluate the methodologies of proposed mapping. The result of the experiment demonstrated that overall on-chip communication cost was reduced by the thread to the core mapping on an average of 12% and up to 26%. The thread only mapping does not improve the on-chip communication cost much; on the other hand; the improvements are significant when thread mapping is combined with the virtual page to DRAM slice mapping. Also, virtual page to DRAM slice mapping is more effective to reduce the overall traffic communication.

2.2 Thread Migration

Migration of thread/process enables fault tolerance, mobile computing, data access locality, eased system administration, and dynamic load distribution. Thread migration may be achieved at the user level, kernel level, or application level. (Shim, Lis, Khan, & Devadas, 2014) considered a mechanism, hardware-level thread migration. They argued that the method has the capability to better exploit of shared data locality for NUCA (Non-Uniform Cache Architecture) designs by adequately supplanting multiple round-trip remote cache accesses by fewer migrations. Thread migrations should be

used judiciously because of the high migration cost; therefore, a novel, on-line prediction scheme was proposed, to choose between performing a thread migration at the instruction level or to do a remote access (as in traditional NUCA designs). The results illustrated that for a set of parallel benchmarks, the proposed technique of thread migration predictor recorded an 18% improvement of the performance on average, also at best by 2.3X over the standard NUCA design that only uses remote accesses.

When deploying multi-threaded applications on-chip multiprocessor (CMP) systems the majority of the designed mechanisms for Non-Uniform Cache Architecture (NUCA) were thread-oblivious. A novel NUCA design was proposed by (Li, Li, Xue, Ouyang, & Shen, 2017) called (CARM), that stands for thread Criticality Assisted Replication and Migration. This approach divides the entire chip of CMP into districts and then take advantage of thread's criticality runtime information. The thread's criticality runtime information is used by the CARM as hints for block replication and migration adjustment in NUCA. The main aim of CARM is boosting the execution of the parallel application by giving the critical thread's block replication and migration a priority over the non-critical thread. The experiments were performed by using Simics, that is a full-system simulator which is execution-driven running Solaris OS on SPARC cores. According to the experimental results, the execution time of a set of PARSEC workloads is reduced by 13.7% compared with D-NUCA and 6.8% on average for Re-NUCA. Furthermore, the energy consumption of CARM much less comparing with the evaluated schemes.

2.3 Optimization Technique

The only level that the Performance Monitoring Counters (PMC)s can be accessed directly is the OS privilege level. For enabling the user-space and the end-user to access PMCs, tools for kernel level access had to be created. To address this inadequacy (Saez, Pousa, Rodriguez-Rodriguez, Castro, & Prieto-Matias, 2017) developed PMCTrack. PMCTrack is a novel tool that equips a simple mechanism for architecture-independent for the Linux kernel that makes accessing the per-thread PMC data possible for the OS scheduler. Although PMCTrack is a tool which is OS-oriented, yet it allows the monitoring data to be collected from userspace, which allows the developers of the kernel to do some necessary debugging and offline analysis to help the developers during the process of designing the scheduler. Likewise, the device gives both the OS and the user-pace PMCTrack segments with otherwise measurements accessible in present-day processors and which are not exposed straightly as PMCs, such as occupancy of cache or consumption of energy.

With the aid of three case studies, they have shown the flexibility and effectiveness of PMCTrack on various range models and architectures of the processor. Three case studies were analyzed on real multicore hardware, the three case studies were: 1) scheduling on asymmetric single-ISA multicore systems 2) measuring power and energy consumption 3) cache monitoring. This data is likewise of extraordinary esteem with regards to breaking down the potential advantages of novel policies of scheduling on real frameworks. The researchers analyzed various case studies that demonstrated the simplicity, powerful features, and flexibility of PMCTrack.

(Chen, Der Bruggen, & Chen, 2018) proposed to use; as an additional option on the task level; the Mixed Redundant Threading (MRT) which is a combination of Simultaneous Redundant Threading (SRT) and Chip-level Redundant Multithreading (CRT). In the coarse-grained approach, the researchers considered on the system level SRT, CRT, and MRT simultaneously, meanwhile the existent results apply only SRT or CRT on the system level, not simultaneously. For more system reliability optimization, two approaches for reliability optimization were proposed to reduce the penalty of system reliability in different degrees, in the time of scheduling the hard real-time tasks on multi-cores. From the embedded benchmark MiBench they chose seven tasks. According to the result of the simulation; compared to the up-to-date techniques; the system reliability is significantly improved.

Methodologies for software synthesis is provided to designers of a real-time embedded system to exploit the techniques of mixed redundancy effectively so that the penalty of system reliability is decreased. In meantime, the timing constraints are satisfied in multi-core systems. Existing state-of-the-art models for user-level tasking and threading either is too specific to or architectures or applications, or are not as flexible or powerful. (Seo et al., 2018) presented Argobots, low-level threading, low-level tasking and a lightweight framework, which was designed as a performant substrate and portable for high-level programming models. Argobots offers an execution model that is carefully designed for balancing generality of functionality and allowing specialization by high-level programming models or end users by providing a rich set of controls. The system was implemented using the C language. For all experiments, the researchers used a 36-core (72 hardware threads) machine, which has two Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of memory. According to the experiment results and evaluation:

1. Besides having richer capabilities Agrobots is competitive with existing threading runtimes.
2. The production OpenMP runtimes have less efficient interoperability capabilities, latency hiding opportunities, and synchronization-reducing than the proposed OpenMP runtime.
3. When MPI interoperates with Argobots instead of Pthreads, it enjoys reduced synchronization costs and better latency-hiding capabilities.
4. An I/O benefit with Argobots can oversee equipment assets all the more effective and diminish obstruction with collocated applications superior to do such an administration with Pthreads.

2.4 Memory

Transactional memory programming paradigm is firstly proposed (Herlihy, Eliot, & Moss, 1993) for replacing locks in concurrent programming. (N Zhou, Delaval, Robu, Rutten, & Méhaut, 2016), by using a Software Transactional Memory (STM) system, investigated thread mapping regulation and autonomic parallelism adaptation. In a parallel program, there are two techniques for gathering the information of the application profile. The first technique is that a master thread can be used for recording the interesting information about itself. The second technique is to use all threads to collect the information.

For the first method, a little synchronization is required to collect information, neither less, the collected information may not symbolize the whole view. The latter technique symbolizes the whole view from the collected information but it's more costly regarding the synchronization cost. The second technique was selected. To collect the profile information they implemented a monitor, the monitor also controls the race condition and the dynamic parallelism. The implementation of the monitor does not require any alteration to the applications. Three parameters were measured from the STM system, namely the physical time, the number of commits and the number of aborts. Commits and aborts were the main variables of the monitor. There were three entry points of the monitor: threads initialization, during a transaction committing and during a thread exiting.

The performance of various static parallelisms was examined and the conclusion was that runtime regulation of parallelism and thread mapping is necessary for the performance of STM systems. The two models outperformed most of the static thread number regarding the performance. The dynamic thread control model shows positive performance rise against the dynamic parallelism model on applications: EigenBench, Yada, and Intruder, but it indicates performance degradation on genome and vacation. If cache will be shared the downside will appear which the competition among the requests from various applications for cache resources, therefore over isolated execution, the execution time will be increased of each application.

Fair-Progress Cache Partitioning (FPCP) has been suggested by (Vicent Selfa, Sahuquillo, Petit, & Gómez, 2017), this approach is a low-overhead cache partitioning hardware-based which can identify system fairness. By allocating; for all applications; a cache partition, the interference can be decreased with FPCP and modify the partition sizes at runtime. This approach estimates modification partitions during multicore execution when any application would have taken in isolation. To estimate the execution time, the auxiliary circuitry is used by FPCP for any application would have experienced if that application is executed without co-runners. This approach attempts to reduce system unfairness by narrowing the progress differences among co-executing tasks. They used SPEC CPU2006 benchmark suite, NAS Parallel Benchmark suite (single-threaded runs) and the ref input set to run the experiments. According to the result of the experiment, without harming the performance, the FPCP approach reduces unfairness by 48% compared to ASM-Cache, in four-application workloads and when using the eight-application workloads the unfairness is reduced by 28%.

(Lu, Yan, Zhou, Zhou, & Zeng, 2017) offered for parallel programming; a new transaction memory model. In the multi-thread paradigm, the data are conflicts. The transaction of data firstly has to be aborted and secondly must be rolled back, and finally should be executed repeatedly till the transaction commits successfully. But in the new approach, the transaction will be appended to the task queue's tail when it aborts N times due to data conflict. The researchers used the C++ language to implement the suggested N-retry software transaction memory (STM).

On the other hand, computers with an Intel i7-6700 processor were used to run the hardware transaction memory (HTM) that had 4 cores. The results of the experiment demonstrated the offered transaction memory model improved the parallel performance; on software transaction memory platform; by 25% and on hardware transaction memory platform the improvement was by 11%. The result also showed a 40% reduction on transaction aborts. meanwhile, up to nowadays, there have not been any processor that gives direct support for Thread-Level Speculation (TLS). Hardware backing for TLS ought to have four key features: (a) detecting data conflict; (b) transactions which are ordered; (c) storage of speculative type; and (d) roll-back in case of detecting a conflict. The IBM POWER8 and Intel Core Hardware Transaction Memory (HTM) support three of these features, so they have the ability to be utilized for supporting the TLS.

As to that, a comprehensive study of the implementation of HTM that supports the loop parallelization with TLS and characterize a careful evaluation of TLS's implementation on the HTM supplements which are offered in such machines have been presented by (Juan Salamanca, Amaral, & Araujo, 2018). The benchmark suites that were used for implementation were the SPEC CPU 2006 and the Collective Benchmark (cBench) which are running on IBM POWER8 and Intel Core. The results indicate that by initializing TLS on top of HTM, some loops may have up to 3.8x speed-ups. The experimental results also revealed that false dependencies may effectively be eliminated by the privatization of memory writes within transactions which are also capable of enabling performance gains with Thread-Level Speculation.

3. DISCUSSION

Table 1 provides an overview of the different systems explained in section II for measuring and even enhancing the operating system performance. As illustrated in the table there is a number of novel approaches been introduced as in (Li et al., 2017), (Saez et al., 2017), (Seo et al., 2018), (V Selfa, Sahuquillo, Petit, & Gómez, 2017) and (Lu et al., 2017) that cover many metrics. The most important approach is Argobots that proposed by (Seo et al., 2018) and includes several features; low-level threading, low-level tasking, and a lightweight framework. This approach depends on a rich set of controls to provide an execution framework which is carefully designed for balancing the generality of functionality and allowing specialization by high-level programming models or end users. Another important approach is the N-retry software transaction memory presented by (Lu et al., 2017). In this approach, the transaction will be appended to the task queue's tail when it aborts N times due to data conflict. The results of this approach illustrate an improvement of 25% and 11% for STM and HTM respectively.

Adding to the above approaches, there are other enhanced approaches (as models) have been produced and work as enhancements of existing models, these approaches proposed by (Ju et al., 2015), (Petrucci et al., 2015a), (Connors & Qasem, 2017), (Pandey & Sahu, 2018), (Shim et al., 2014), (Chen et al., 2018), (N Zhou et al., 2016), and (J Salamanca, Amaral, & Araujo, 2018). Considering thread mapping metric, the optimization model of Integer Linear Programming (ILP) proposed by (Petrucci et al., 2015b) outperforms other proposed thread assignment schemes that do not address memory transfer speed requirements. Benefiting from the capabilities of Linux scheduling and performance-monitoring, the prototype of this model was implemented and evaluated at the user level.

According to the simulation results, this approach accomplishes huge performance gains and energy savings. As for memory enhancement, (Juan Salamanca et al., 2018) illustrated that by initializing TLS on top of HTM, some TLS loops may have up to 3.8x speed-ups. Also revealed that false dependencies may effectively be eliminated by the privatization of memory writes within transactions, which are also capable of enabling performance gains with Thread-Level Speculation.

Table 1. Summary

Ref.	Depended Features				
	Year	*Metric*	*Novelty*	*Used hardware/software*	*Result*
(Ju et al., 2015)	2015	Thread Mapping		IXP1200/2400 workbenches	The maximal sustainable line rates estimated using the simulation tool 6% and it gets up to 8% by using the queuing network of cycle-accurate
(Petrucci et al., 2015a)	2015	An optimization model of ILP and a scheme to set the thread-to-core assignment dynamically.		SPEC benchmark programs	. The result showed that EDP gains of average a 15% to 35% maximum for a variety of workloads
(Connors & Qasem, 2017)	2017	Driving the tasks of feature extraction, feature selection, training data generation, model training, evaluation, and selection		Nvidia Tesla K40c GPU on a Linux system that had CUDA 7.5 installed.	They found that different block sizes are needed due to subtle differences in a kernel's runtime behavior
(Pandey & Sahu, 2018)	2017	Virtual page to memory slice mapping		Sniper simulator (version 6.1)	The overall on-chip communication cost was reduced by the thread to core mapping on an average of 12% and up to 26%.
(Shim et al., 2014)	2014	A mechanism, hardware-level thread migration		Pin and Graphite to model the proposed NUCA architecture	Predictor recorded an 18% improvement of the performance on average and at best by 2.3X over the standard NUCA
(Li et al., 2017)	2017	Runtime thread criticality information	thread Criticality Assisted Replication and Migration (CARM)	Simics, a set of PARSEC workloads	Execution time is reduced by 13.7% and 6.8% on average for Re-NUCA. Furthermore, the energy consumption of CARM much less compared with the evaluated schemes.
(Saez et al., 2017)	2017	Accessing the per-thread PMC data for the OS scheduler	PMCTrack	Linux kernel (v4.1.5) equipped	Simplicity, flexibility and powerful features of PMCTrack.
(Chen et al., 2018)	2018	Decreasing system reliability penalty		Benchmark MiBench	The result of simulation; compared to the state-of-the-art techniques; showed that the system reliability is significantly improved
(Seo et al., 2018)	2018	Low-level threading, low-level tasking and a lightweight framework	Argobots	A 36-core (72 hardware threads) machine, which has two Intel Xeon E5-2699 v3 (2.30 GHz) CPUs and 128 GB of memory.	Richer capabilities, reducing latency, reduced synchronization costs and better latency-hiding capabilities
(Naweiluo Zhou, Delaval, Robu, Rutten, & Mehaut, 2016)	2016	Thread mapping regulation and autonomic parallelism adaptation		EigenBench, yada and intruder	The two models outperformed most of the static thread number regarding the performance.
(V Selfa et al., 2017)	2017	Cache partitioning	Fair-Progress Cache Partitioning (FPCP)	SPEC CPU2006 benchmark suite, NAS Parallel Benchmark suite (single-threaded runs)	Reduces unfairness by 48% compared to ASM-Cache, in four-application workloads and when using the eight-application workloads the unfairness is reduced by 28%.
(Lu et al., 2017)	2017	Transaction memory model	N-retry software transaction memory	C++ language, computer with an Intel i7-6700 processor	Improved; on software transaction memory platform; the parallel performance by 25% and on hardware transaction memory platform the improvement was by 11%. The result also showed a 40% reduce on transaction aborts
(Juan Salamanca et al., 2018)	2018	Hardware transaction memory		SPEC CPU 2006 and the Collective Benchmark (cBench) which are running on IBM POWER8 and Intel Core.	Some loops may have up to 3.8x speed-ups. And also alse dependencies may effectively be eliminate by privatization of memory writes within transactions

4. CONCLUSION

From the comparison table illustrated in section 3, it can be concluded that a very active approach with important novelty has been proposed by Sangmin Seo. This approach covered most important metrics includes; low-level threading, low-level tasking, and a lightweight framework. Also, it is suggested to depend on models such as that produced by Petrucci and Salamanca to measure and improve the performance of the addressed operating systems and privatization of memory writes within transactions in order to accomplish huge performance gains and energy savings.

REFERENCES

Chen, K. H., Der Bruggen, G. Von, & Chen, J. J. (2018). Reliability Optimization on Multi-Core Systems with Multi-Tasking and Redundant Multi-Threading. *IEEE Transactions on Computers*, *67*(4), 484–497.

Connors, T. A., & Qasem, A. (2017). Automatically Selecting Profitable Thread Block Sizes for Accelerated Kernels. *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 442–449.

Herlihy, M., Eliot, J., & Moss, B. (1993). Transactional Memory: Architectural Support For Lock-free Data Structures. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, 289–300.

Jarp, S., Jurga, R., & Nowak, A. (2008). Perfmon2: A leap forward in performance monitoring. *Journal of Physics: Conference Series*, *119*(4), 1–6.

Ju, M., Jung, H., & Che, H. (2015). A Performance Analysis Methodology for Multicore, Multithreaded Processors. *IEEE TRANSACTIONS ON COMPUTERS*, *63*(2), 276–289.

Li, J., Li, M., Xue, C. J., Ouyang, Y., & Shen, F. (2017). Thread criticality assisted replication and migration for chip multiprocessor caches. *IEEE Transactions on Computers*, *66*(10), 1747–1762.

Lu, K., Yan, C., Zhou, H., Zhou, D., & Zeng, X. (2017). A Novel N-Retry Transactional Memory Model for Multi-Thread Programming. *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, 814–821.

Monitor (synchronization). (2018). In *Wikipedia*. Retrieved from https://en.wikipedia.org/w/index.php?title=Monitor_(synchronization)&oldid=860252631

Musiphil. (n.d.). Monitor. Retrieved from https://en.wikipedia.org/wiki/Monitor_(synchronization)

Pandey, R., & Sahu, A. (2018). Efficient Mapping of Multi-threaded Applications onto 3D Stacked Chip-Multiprocessor. *Proceedings - 2017 IEEE 19th Intl Conference on High Performance Computing and Communications, HPCC 2017, IEEE 15th Intl Conference on Smart City, IEEE 3rd Intl Conference on Data Science and Systems,* 324–331.

Petrucci, V., Loques, O., Mossé, D., Melhem, R., Gazala, N. A., & Gobriel, S. (2015a). Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems. *ACM Transactions on Embedded Computing Systems*, *14*(1), 1–26.

Petrucci, V., Loques, O., Mossé, D., Melhem, R., Gazala, N. A., & Gobriel, S. (2015b). Energy-Efficient Thread Assignment Optimization for Heterogeneous Multicore Systems. *ACM Trans. Embed. Comput. Syst.*, *14*(1), 15:1--15:26.

Rashid, Z. N., Sharif, K. H., & Zeebaree, S. (2018). Client / Servers Clustering Effects on CPU Execution-Time , CPU Usage and CPU Idle Depending on Activities of Parallel-Processing-Technique Operations ". *INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH*, *7*(8), 106–111.

Saez, J. C., Pousa, A., Rodriguez-Rodriguez, R., Castro, F., & Prieto-Matias, M. (2017). PMCTrack: Delivering performance monitoring counter support to the OS scheduler. *Computer Journal*, *60*(1), 60–85.

Salamanca, J, Amaral, J. N., & Araujo, G. (2018). Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation. *IEEE Transactions on Parallel and Distributed Systems*, *29*(2), 466–480.

Salamanca, Juan, Amaral, J. N., & Araujo, G. (2018). Using Hardware-Transactional-Memory Support to Implement Thread-Level Speculation. *IEEE Transactions on Parallel and Distributed Systems*, *29*(2), 466–480.

Selfa, V, Sahuquillo, J., Petit, S., & Gómez, M. E. (2017). A Hardware Approach to Fairly Balance the Inter-Thread Interference in Shared Caches. *IEEE Transactions on Parallel and Distributed Systems*, *28*(11), 3021–3032.

Selfa, Vicent, Sahuquillo, J., Petit, S., & Gómez, M. E. (2017). A Hardware Approach to Fairly Balance the Inter-Thread Interference in Shared Caches. *IEEE Transactions on Parallel and Distributed Systems*, *28*(11), 3021–3032.

Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., … Beckman, P. (2018). Argobots: A Lightweight Low-Level Threading and Tasking Framework. *IEEE Transactions on Parallel and Distributed Systems*, *29*(3), 512–526.

Shim, K. S., Lis, M., Khan, O., & Devadas, S. (2014). Judicious Thread Migration When Accessing Distributed Shared Caches. *IEEE Computer Architecture Letters*, *13*(1), 53–56.

Zeebaree, S. R. M., & Jacksi, K. (2015). Effects of Processes Forcing on CPU and Total Execution-Time Using Multiprocessor Shared Memory System. *INTERNATIONAL JOURNAL OF COMPUTER ENGINEERING IN RESEARCH TRENDS*, *2*(4), 275–279.

Zhou, N, Delaval, G., Robu, B., Rutten, É., & Méhaut, J. (2016). Autonomic Parallelism and Thread Mapping Control on Software Transactional Memory. In *2016 IEEE International Conference on Autonomic Computing (ICAC)* (pp. 189–198).

Zhou, Naweiluo, Delaval, G., Robu, B., Rutten, E., & Mehaut, J. F. (2016). Autonomic parallelism and thread mapping control on software transactional memory. In *Proceedings - 2016 IEEE International Conference on Autonomic Computing, ICAC 2016* (pp. 189–198).