STAND: New Tool for Performance Estimation of the Block Data Processing Algorithms in High-load Systems

Victor Minchenkov, Vladimir Bashun
St-Petersburg State University of
Aerospace Instrumentation
{victor, bashun}@vu.spb.ru

Alexander Povalyaev
EMC St-Petersburg
Development Center
alexander.povalyaev@emc.com

Abstract

The main goal of this work is to present the developed research tool to find, investigate and analyze hidden dependences between parameters of the hardware/software platforms (such as influence of NUMA architecture, memory page size, etc) and the performance of block data processing algorithms. The new toolset (STAND) allows performance estimation and comparison of block data processing algorithms (for example, encryption/compression algorithms) running in kernel space. The primary application area of the developed technology and toolset is performance estimation and comparison of "black box" libraries on particular hardware/software platform rather than research of mathematical or software implementation of algorithms. The main advantage of the presented toolset is that no source codes of algorithm implementation are needed (providing that an abstraction layer with known API is available). Linux operating system and computing nodes with ccNUMA architecture was selected as basic software/hardware platform. In this paper, the architecture of STAND is described. The methods for generating system load and comparison results for encryption algorithms AES (CBC), AES (CTR), and compression algorithms LZO, quicklz and bCodec are also presented.

Index Terms: Algorithms, Performance analysis, Block data processing, ccNUMA.

I. Introduction

There exists a variety of programs for Linux operating system which allow estimating the performance of different modules of system kernel, or performance of system hardware. For example, the Flexible IO Tester utility [1] can be used to benchmark and stress test I/O system and estimate the performance of working with disks. Geekbench - cross-platform benchmark [2] provides a set of benchmarks to measure memory and processor (work with integer and floating point) performance. SysBench - system performance benchmark [3] tool for evaluating different Linux OS parameters (scheduler, memory allocation, file I/O). All these systems use its own set of specific tests and methods for generating system load. In most cases, these tests are static and do not have much adjustable parameters.

The main goal of the developed tool (called STAND), on contrary, is estimation of not hardware/software platform performance itself, but measurement of performance of block data processing algorithms on the platform. The examples of such algorithms are compression and encryption algorithms. Such algorithms are widely used in modern information storage and management systems. In such systems data is not just "stored", but additionally processed (e.g. compressed and/or encrypted). Processing is usually done inside low-level services, in kernel space. For such systems the performance of algorithms

under high load becomes crucial factor. Here arises task of comparing and performance estimation of different block-data processing algorithms.

The assumption is that we can modify neither the algorithms, nor their implementations. Yet we want to compare and estimate performance of such algorithms. Even more, we can try to optimize the parameters of environment to improve the performance of algorithms.

The utility may have no details of particular algorithm implementation, provided that an abstraction layer with known API is available. The subject of research is performance estimation and comparison of “black box” libraries (running in kernel space) on particular hardware/software platform.

To emulate the influence of real-world environment, several emulators of various system loads were also developed.

The work is organized as follows: firstly, we present and explain the STAND architecture and parameters. Then experiment results for some compression and encryption algorithms are presented. The conclusion is presented in the last part.

II. STAND PERFORMANCE ESTIMATION UTILITY

A. STAND components

STAND analyzes performance of block data processing algorithms, running in kernel space. It satisfies the following requirements:

- It is easy to adopt new algorithms inside STAND as libraries (even if their source code is not available).
- User can set the plan for experiment using a simple configuration file.
- Running STAND is as easy as calling executable file with configuration file for experiment as parameter.

STAND architecture consists of three main parts (see Fig. 1):

1) **User interface** – used by operator to interact with part of STAND inside OS kernel (kernel driver). It is implemented as executable file. Main tasks are: sending parameters of experiment to kernel, receiving the results.

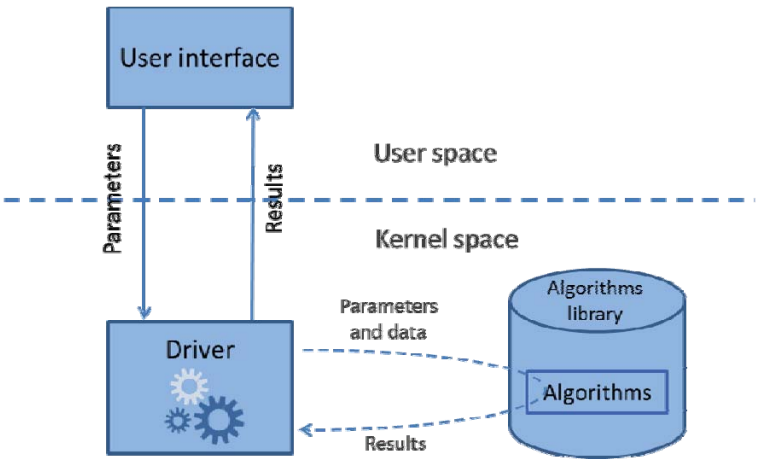


Fig. 1. STAND components

2) **Kernel driver** – main part of STAND. Its tasks are: starting algorithm, gathering statistics, controlling system load. It's in turn can be split to (see Fig. 2):
   a. *Control module* – coordinates work of all other driver modules. Sends results of experiment back to user interface.
   b. *Profiling module* – measures execution time for algorithms.
   c. *System load module* – creates system load, emulating "real" environment.
   d. *Memory pool* – used to ease memory allocation on given NUMA node. Memory, allocated by the algorithm, is taken from this pool.
3) **Algorithms library** – stores all tested algorithms.

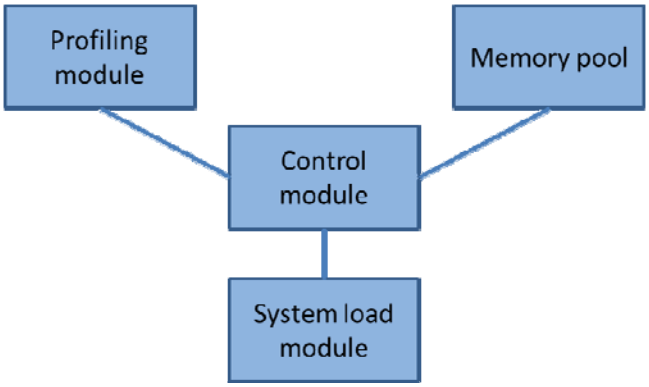


Fig.2. Kernel driver components

To adopt tested algorithm to STAND, one need to create interface for connection of this algorithm. Each block data processing algorithm works with following memory: memory for input data, memory for results, local buffer (stack). The following function, providing connection of library to STAND, should be defined:

*static noinline int foo(char *inp_mem, int inp_mem_size, char *outp_mem, int outp_mem_size, int pull_mem_node, allocator pull_malloc, char *error)*;

where:

- *inp_mem* – pointer to input data;
- *inp_mem_size* – input data size;
- *outp_mem* – pointer to buffer, where output data (results) should be written;
- *outp_mem_size* – output buffer size;
- *pull_mem_node* – NUMA node where algorithm may allocate data (for stack);
- *pull_malloc* – pointer to dynamic memory allocation function having the following prototype:
- *void *allocator(unsigned int size, unsigned int node),*
- where:
  - *size* – size of allocated memory;
  - *node* – NUMA node, where memory shall be allocated;
- *error* – buffer, used by algorithm to write error message (256 bytes max).

The function must return number of bytes, written to output buffer, -1 on error.

The STAND profiling module measures algorithms execution time only, so overhead of STAND functions does not influence the measured results.

B. STAND parameters

Time needed to process block of data shall be estimated. This is the key parameter for all storage systems.

All selected algorithms are block data processing algorithms, and size of data block can be varied. Hence, block size can be one of algorithms parameters. Size of block is one of the factors that may influence the algorithm performance.

Various factors may influence the environment during processing. It is hard to take into consideration all the factors. That is why the STAND is scalable, its architecture allows introduction of new factors. At this moment, the following factors are used: additional load on CPU, additional load on TLB cache and additional load on QPI link.

1) CPU load: System load emulation is implemented using calculations with integers, without work with memory. The following parameters of CPU load that can be set using configuration file: number of threads (the intensity of the load), location of load (user-space or kernel), number of core (the core where the load process will run).

2) TLB cache load: Translation lookaside buffer (TLB) is a specialized cache of central processor used to improve virtual address translation speed. Every record of TLB contains mapping from virtual to physical memory address. If there is no record in TLB (TLB miss), address translation should be done, which takes up to 10-60 times longer. Average TLB miss probability is 0.01%-1% [4]. This rate can be estimated by profiler tools like oProfile. TLB load emulator attempted to address memory in such way that it should increase the probability of TLB miss. Parameters of the TLB load are: number of core to run on, memory block size for write (taken as TLB size multiplied by memory page size), number of NUMA node (should be the same NUMA node for TLB load and tested algorithm).

3) QPI link load: To understand the process of loading QPI link, we need to give some additional comments of ccNUMA architecture. ccNUMA (cache coherency Non-Uniform Memory Access) is an architecture, where the memory access time depends on the memory location relative to a processor [5]. Processors (cores) are united in NUMA nodes. Under NUMA, a processor can access its own local memory (memory of its NUMA node) faster than non-local memory (memory of other NUMA nodes). At the same time, access to non-local memory increases load interconnect link between processors. Intel processors use QuickPath Interconnect (QPI) links as a communication medium for ccNUMA implementation. As shown in [6], usage of non-local memory may decrease performance up to 30%. At [7] comparable results shown. These results were obtained for special tests, and may be different for real algorithms. STAND allows estimation of this factor for incorporated algorithms.

Using non-local memory may decrease performance itself. Additional load on QPI link may make strengthen this factor. Additional load means that not only algorithm work with non-local memory, but load processes generate traffic on QPI link also. To emulate load on QPI link, load process should actively work with non-local memory.

Parameters of the QPI link load are: number of core to run, memory size, location of emulator (user-space or kernel), NUMA node number (for non-local memory).

III. PRACTICAL RESULTS

A. Experiment description

Experiments were conducted on servers with Intel XEON E5620 processors, supporting NUMA architecture (Nehalem).

The following parameters of tested platform were also known:

- Linux kernel version 2.6.32-71.el6.x86_64;
- Two quad-core processors Intel Xeon E5620 2.40 GHz;
- Hyper threading is on;
- Total number of cores – 16;
- Number of numa nodes – 2.

The following block data processing algorithms were tested: encryption algorithms AES (CBC) and AES (CTR) from open source cryptographic library OpenSSL [8], and open-source implementations of compression algorithms LZO [9] and quicklz [10]. As a proof of "black box" library concept, an implementation of compression algorithms bCodec with closed sources but known API interface was used.

One of the assumptions was that block size may influence the algorithm performance. Hence, set of experiments with different sizes of block were performed. Block sizes 4Kb, 8Kb, 16Kb, 32Kb and 64 Kb were taken.

Another goal was to investigate how additional system load influence the algorithm performance. On each experiment one algorithm was taken with a set of additional loads (CPU, TLB or QPI).

Number of iterations for each experiment was $2*10^5$, and total size of processed data was $2*2^{30}$ bytes. Time measurements were taken for every processed block separately.

B. Experiment results

Experiments results are presented below. Block sizes used during processing are plotted on the abscissa axis. Time spent to process 4Kb block is plotted on the ordinate axes. Four curves are plotted on each figure: 'min' (minimal time needed to process block of data throughout experiment), 'max' (maximum time needed to process block of data throughout experiment), 'avg' (average time needed to process block of data throughout experiment) and 'no load' (average time for algorithm, with no additional system load). Comparison of 'avg' and 'no load' allows evaluation of system load impact on algorithm performance. Curves 'min' and 'max' gives some insight on results dispersion.

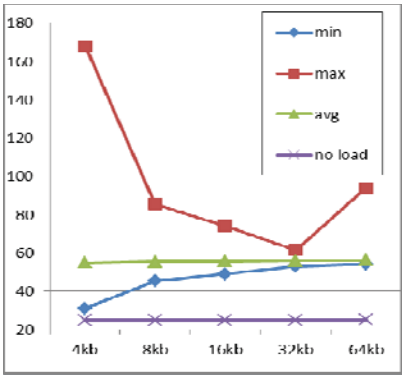


Fig. 3. AES-CBC, CPU load

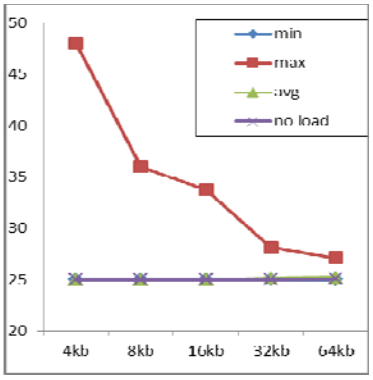


Fig. 4. AES-CBC, TLB load

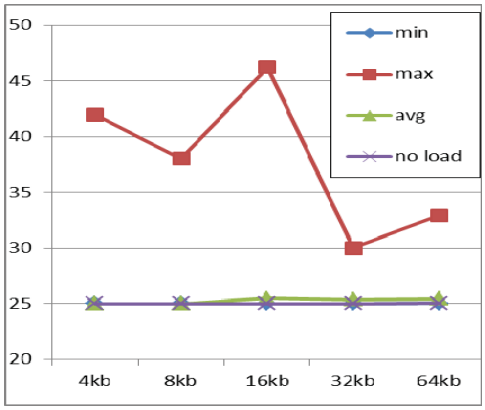


Fig. 5. AES-CBC, QPI link load

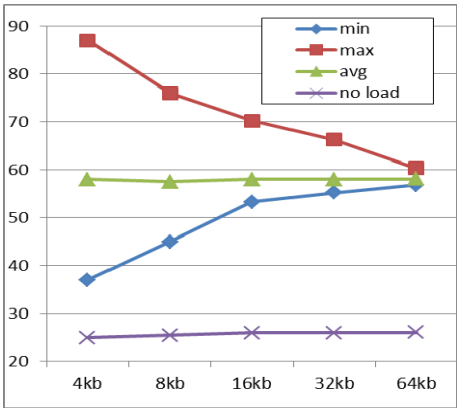


Fig. 6. AES-CTR, CPU load

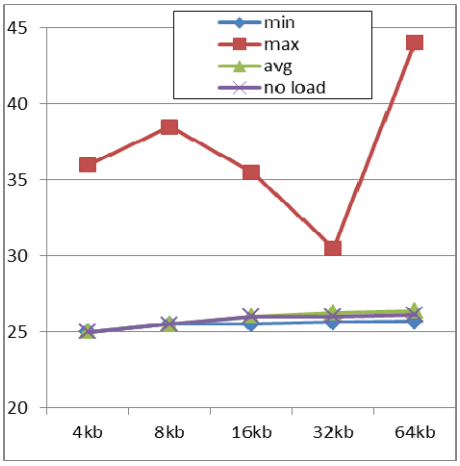


Fig. 7. AES-CTR, TLB load

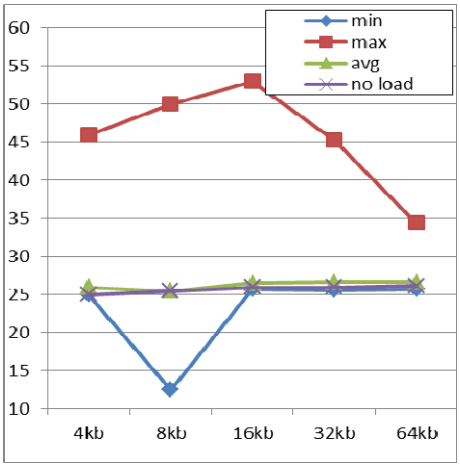


Fig. 8. AES-CTR, QPI link load

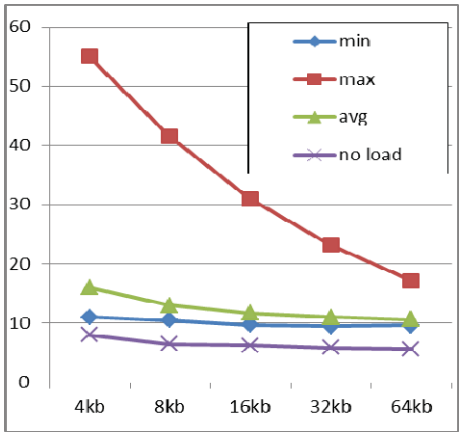


Fig. 9. quicklz, CPU load

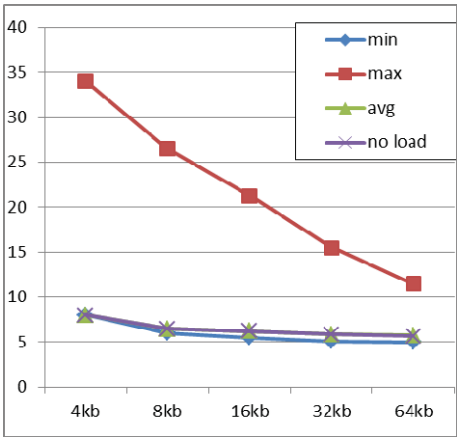


Fig. 10. quicklz, TLB load

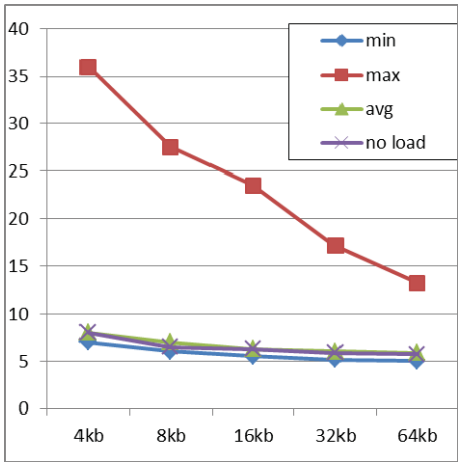


Fig. 11. quicklz, QPI link load

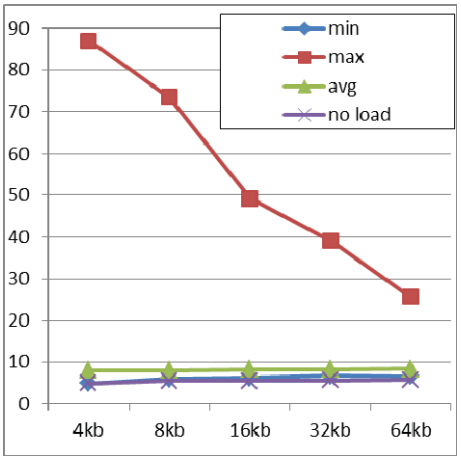


Fig. 12. LZO, CPU load

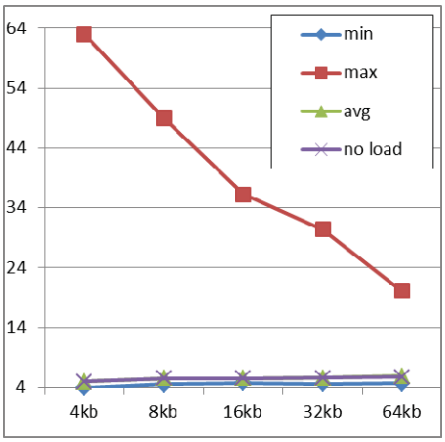


Fig. 14. LZO, TLB load

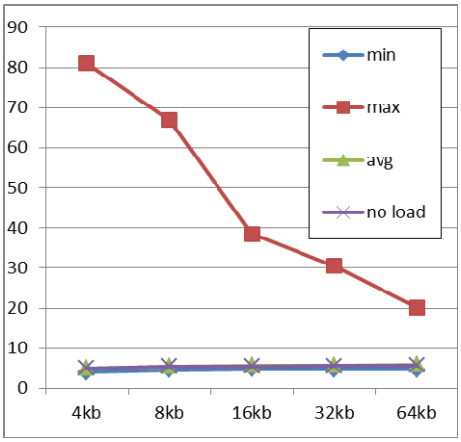


Fig. 15. LZO, QPI link load

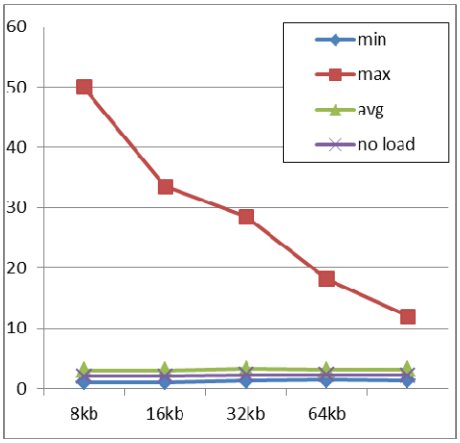


Fig. 16. bCodec, CPU load

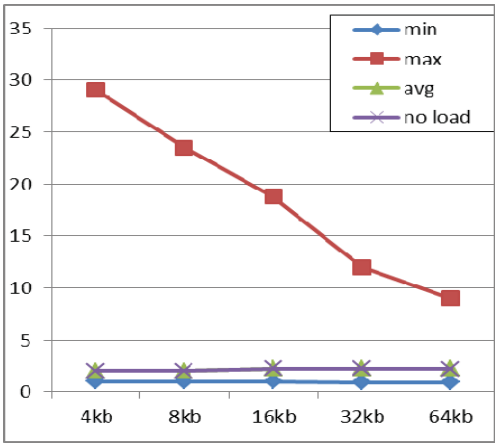


Fig. 17. bCodec, TLB load

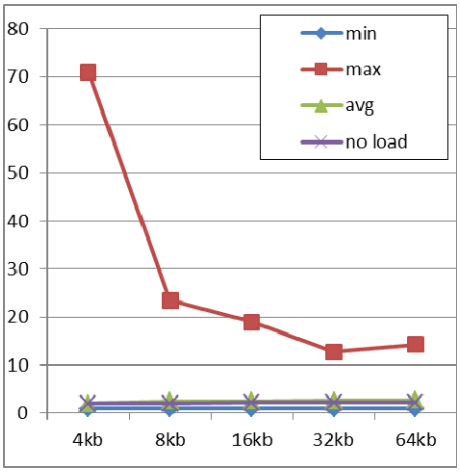


Fig. 18. bCodec, QPI link load

From the results of experiments the following can be seen:

1) Increasing of **block data size** was positive in terms of performance only for quicklz algorithm. The reason for this may be in the way the algorithm works with memory. We assumed to see similar behavior for LZO, but it was not (5% degradation). All other algorithms also showed performance degradation with increasing of block size.
2) **CPU load** shows the most influence on performance. This may be explained as follows. The system load was set on the same physical core with tested algorithm, so performance degradation if determined by hyper threading technology. On algorithms bCodec and LZO it influenced less (50% degradation), other algorithms showed almost 100% performance degradation. This can be partly explained by the fact that for bCodec and LZO, one iteration ('no load') takes from 2 to 10 times less than for other algorithms.
3) **TLB load** showed much less influence on performance. E.g. for compression algorithms it caused 2% degradation on 64Kb block. Additional analysis showed that accuracy of developed TLB load may be insufficient, and further clarification of computing node model is needed.
4) **QPI link load** showed noticeable influence on performance (2%) only on 64Kb block. We noticed that **QPI link load and TLB load** cause performance degradation only for algorithms that work actively with memory, and the more algorithm works with memory, the higher will be degradation. STAND allows estimation of this impact on particular algorithm. For selected algorithms, performance degradation from ccNUMA non-local memory factor was far from maximum 30% decrease met in literature.
5) Additional data was gathered to estimate the impact of max time. On Fig. 19 a distribution of time measures for LZO algorithm with TLB load and block size 64 Kbytes is presented. One can see that maximum time ('max' curve) can be seen in sporadic cases. Yet, for real time systems and systems providing some quality of service, such cases should also be taken into account.

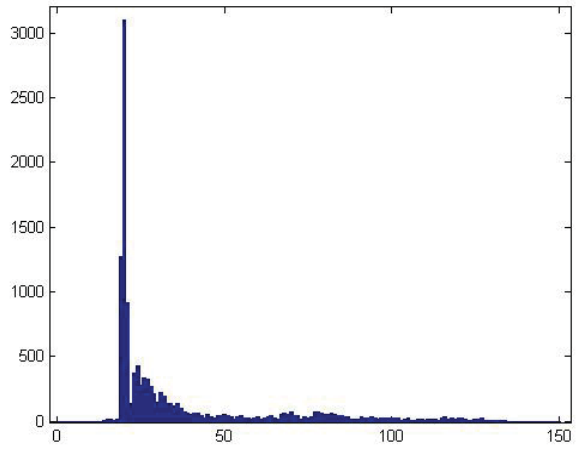
6) 

Fig.19. distribution of execution time, LZO with TLB load, block 64 Kbytes

IV. Conclusion

A new tool (STAND) for performance estimation and comparison of “black box” libraries on particular hardware/software platform was developed. STAND analyzes performance of block data processing algorithms, running in kernel space. Such algorithms are used in modern information storage and management systems.

STAND have simple interface for inclusion of new libraries with block data processing algorithms. Any algorithm implementation can be included to STAND, providing that an abstraction layer with described API is available or can be written. Several known and widely used implementations compression and encryption algorithms were introduced to STAND to prove this concept.

A model of computing node was selected based on information from open literature. Three types of system load were developed in accordance to selected model. The architecture of STAND allows increasing number of parameters in future.

A set of experiments with widely used block data processing algorithms (compression and encryption) was done. Conducted experiments confirmed the possibility to use STAND to estimate the influence of various factors on algorithms (implemented in particular libraries) performance. Experiment results allowed making some practical conclusions. For example, an optimal block size for given high loaded system can be chosen.

Inclusion and analysis of new libraries may be one direction of future work. Improvement of model of computing node and appropriate modification of STAND is another. Introducing additional factors of environment (memory page size, scheduler type, use of virtualization, etc.) as new parameters of STAND is the third direction.

Acknowledgment

The authors would like to thank EMC Corporation for supplied hardware.

References

[1] Flexible IO Tester, http://www.obsecurities.com/reviews/5-fio-flexible-io-tester-review.
[2] Geekbench - cross-platform benchmark, http://www.primatelabs.ca/geekbench/.
[3] SysBench - system performance benchmark, http://sysbench.sourceforge.net/.

[4] David A. Patterson, John L. Hennessy, “Computer Organization And Design. Hardware/Software interface. 4th edition”, *Burlington, MA 01803, USA: Morgan Kaufmann Publishers*, 2009.
[5] Christoph Lameter, “Local and Remote Memory: Memory in a Linux/NUMA System”, Jun 20th, 2006.
[6] Iakovos Panourgias, “NUMA effects on multicore, multi socket systems”, *The University of Edinburgh*, 2011.
[7] Kayi, Abdullah et al., “Application Performance Tuning for Clusters with ccNUMA Nodes”, *Computational Science and Engineering*, 2008. CSE '08.
[8] OpenSSL, http://openssl.org/.
[9] LZO, http://www.oberhumer.com/opensource/lzo/.
[10] Quicklz, http://www.quicklz.com/.