Chapter 21
Analysis and Optimization of CFS Scheduler on NUMA-Based Systems

Hongyun Tian, Kun Zhang, Li Ruan, Mingfa Zhu, Limin Xiao, Xiuqiao Li and Yuhang Liu

Abstract Non Uniform Memory Access (NUMA) architecture becomes more and more popular as it has better scalability than Uniform Memory Access (UMA). However, all previous work on the operation system scheduler assumed that the underlying system is UMA. As a result, the performance degrades on NUMA machines due to lacking of consideration to the underlying hardware. Researchers discover that the Completely Fair Scheduler (CFS) does not work smoothly on NUMA machines and even interfere performance relative to the O (1) scheduler. In this paper researchers investigate the causes for the performance decline and devise an architecture aware task-bound approach for NUMA system, which can help the CFS scheduler works efficiently on NUMA platforms. The evaluation shows that the approach can upgrade the system performance by more than 60 % on average. The research has great significance to the development and popularity of domestic operating system.

Keywords NUMA · CFS scheduler · Operating system · High-performance computer

H. Tian (✉) · K. Zhang · L. Ruan (✉) · M. Zhu · L. Xiao · X. Li · Y. Liu
State Key Laboratory of Software Development Environment, Beijing, China
e-mail: sympathyh@gmail.com

L. Ruan
e-mail: ruanli@buaa.edu.cn

H. Tian · L. Ruan
School of Computer Science and Engineering, Beihang University, Beijing, China

W. E. Wong and T. Ma (eds.), *Emerging Technologies for Information Systems, Computing, and Management*, Lecture Notes in Electrical Engineering 236, DOI: 10.1007/978-1-4614-7010-6_21,

21.1 Introduction

UMA architecture has been widely used in kinds of computer architectures. As shown in Fig. 21.1, all cores access to the same memory node according to the bus line. However, the memory access according to the bus line will be sharply increases as the number of cores per processor increases. As a result, bus contention turns to be the bottleneck of the system. New multicore systems increasingly use the NUMA architecture due to its better decentralized and scalable nature than UMA. There are multiple memory nodes in the NUMA systems. Each node has its own memory controller, compute nodes use Hyper Transport Bus to connect with each other. Each core can access to the memory on its own node and other nodes with different access latency, the memory access to the local node (*local access*) can be faster than the remote node (*remote access*). The bus contention is diminished but schedule strategy needs to be carefully decided to avoid remote memory access.

Linux is a leading operating system on servers and other big iron systems [1]. The task scheduler is a key part of Linux operating system and Linux continues to evolve and innovate in this area. A lot of good schedulers have been implemented by the kernel developers. O(1) scheduler and CFS scheduler are two most popular schedulers among them.

The O(1) scheduler is a multi-queue scheduler, each processor has a operation queue, but it cannot detect the node layer on NUMA systems. As a result, it cannot guarantee the process in scheduling keep running on the same node. Therefore, Eirch Focht developed a node affinitive NUMA scheduler based on the O(1) scheduler. But the O (1) scheduler needs large mass of code to calculate heuristics and became unwieldy in the kernel [2]. Ingo Molnar then developed the CFS based on some of the ideas from Kolivas's RSDL scheduler. CFS has been a part of the Linux since kernel 2.6.23. The purpose of CFS is to make sure that all the processes need run time could get an equal and fair share of processing time. It makes a progress in the fairness of assigning runtime among tasks but unfortunately it didn't take the under hardware layer into account. Processes may need to remote

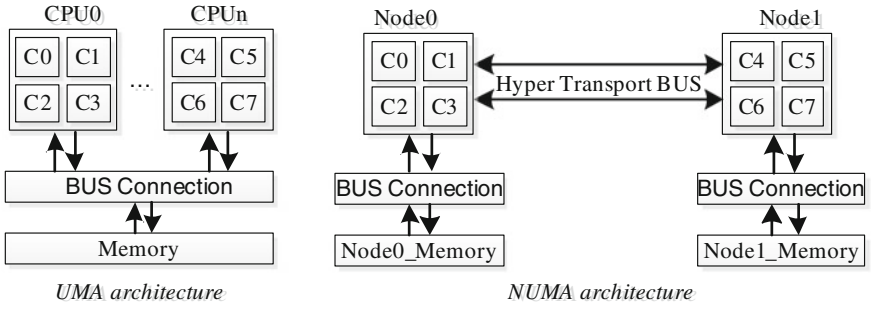


Fig. 21.1 Schematic overview of UMA and NUMA systems

access to its memory frequently, which can cause the performance degrade sharply. As a result, it cannot work well on the NUMA platform compared with the O(1) scheduler with NUMA patch.

We discover that the CFS scheduler not only fails to managing processes effectively on NUMA systems but even hurts performance when compared to the O(1) scheduler with Eirch Focht's NUMA patch. Our experiment setup on an NUMA system based on Loongson CPU, we use LMbench to evaluate the pipe bandwidth and latency, the test score shows that CFS scheduler will degrade as much as 40 % relative to the O(1) scheduler with NUMA patch.

The focus of our study is to investigate why CFS fails to work smoothly on NUMA platforms and devise the architecture aware task-bound approach that would help CFS work efficiently on NUMA platforms. The rest of this paper is organized as follows. Section 21.2 demonstrates why CFS scheduler fails to work well on NUMA systems. Section 21.3 presents our improved measure. Section 21.4 evaluates the task-bound approach. Section 21.5 discusses the related work before we make a conclusion about our research in Sect. 21.6 and present our acknowledgment in last section.

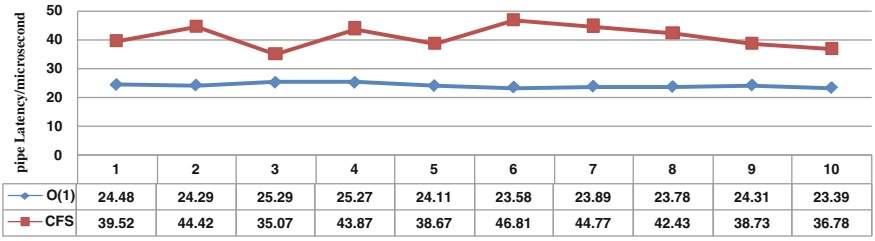
21.2 Motivation

The focus of this section is to experimentally demonstrate why CFS fails to work smoothly on NUMA platforms. We quantify the effects of performance degradation with benchmarks from the LMbench benchmark suite. We perform experiments on a Dual way NUMA server equipped with a Loongson3 processor per node running at 1 GHz, and 4 GB of RAM per node. The kernel of the operating system is Linux 2.6.36.1. Figure. 21.1 schematically represents the architecture of the dual way server.

To quantify the effects of performance degradation caused by the CFS scheduler, we run the bw_pipe (a tool to test the pipe communication bandwidth) and lat_pipe (a tool to test the pipe communication latency) sub items of LMbench to test the pipe communication bandwidth and latency. Besides, we look into the schedule pattern of the benchmark process use the linux command top.

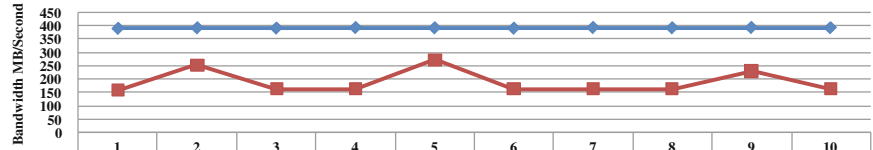
As we depicted in Figs. 21.2 and 21.3, the test scores show that the pipe latency with CFS scheduler grows 67.9 % on average while the bandwidth degrades 51.6 % compared with the NUMA patched O(1) scheduler. That is really bad! Besides, the test scores with CFS scheduler show strong randomness feature while the test scores with O(1) scheduler are far more stable.

To quantify the cause of the big difference between O(1) NUMA scheduler and CFS scheduler, we use top to look into the schedule pattern of the benchmark process during the test. Finally we find out that the test processes were scheduled among the 8 cores randomly by the CFS scheduler, while the O(1) NUMA scheduler always try to let the process running on the same core during the test. These results demonstrate a very important point that the CFS scheduler cannot



	1	2	3	4	5	6	7	8	9	10
O(1)	24.48	24.29	25.29	25.27	24.11	23.58	23.89	23.78	24.31	23.39
CFS	39.52	44.42	35.07	43.87	38.67	46.81	44.77	42.43	38.73	36.78

Fig. 21.2 Contrast test with lat_pipe in LMbench



	1	2	3	4	5	6	7	8	9	10
O(1)	389.27	391.28	390.25	392.18	391.23	389.78	392.47	391.39	392.59	391.69
CFS	158.4	251.38	164.13	164.3	271.47	161.89	164.39	163.75	230.21	164.18

Fig. 21.3 Contrast test with bw_pipe in LMbench

work efficiently due to lacking of consideration of the hardware architecture, the CFS scheduler does not distinguish the cores on the NUMA nodes and just assign tasks randomly to the cores.

Now that we are familiar with causes of performance degradation on NUMA systems, we are ready to explain why CFS scheduler fails to work efficiently on NUMA platforms. The main idea behind the CFS is maintaining balance fairness. To determine the balance, the CFS maintains the amount of time provided to a given task which called the virtual runtime, and the CFS maintains a virtual runtime ordered red-black tree (see Fig. 21.4) rather than run queue as has been done in prior Linux scheduler. The CFS scheduler always choose the process on the most left node of the RB tree and choose a free core to run the process.

Suppose that there are several processes p0, p1, p2, p3, p4, p5, p6, p7, p8, p9 in the system, the orders of the virtual runtime among these processes are $p9 > p8 > p7 > p6 > p5 > p4 > p3 > p2 > p1 > p0$. The CFS scheduler tries to choose the smaller virtual runtime process to run first. As a result, p0 to p7 are assigned to run on the 8 cores while p8 and p9 still in the RB tree waiting to be scheduled. In the next clock interrupt, the p7 finishes its work and then be deleted from the RB tree while the p0 process was moved to the end of the tree as its runtime increases, the p8 and p9 then get the chance to run on the cores. But in the come clock interrupt, another process, take p5 for example, finishes its work and been deleted from the tree. Then at this clock interrupt, the p0 is reassigned to core 5. The problem of remote memory access coming out as p0's memory is on the memory of node 0.

To summarize, CFS scheduler ignores the under layer of the hardware and finally causes the performance degradation. It fails to eliminate remote memory

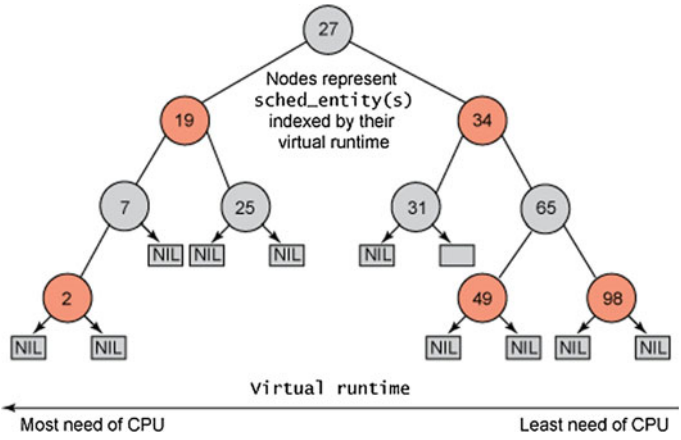


Fig. 21.4 Example of a *red-black* tree [3]

access and even introduces remote latency overhead. To solve this problem, we devise the task-bound approach to avoid remote memory access automatically.

21.3 Implementation

In order to devise the automatic task-bound approach exploiting NUMA architectures, it needs some kind of description to the system. For instance the Advanced Configuration and Power Interface Specification (ACPI), it provides the distance between hardware resources on different NUMA nodes [4]. But the ACPI does not define how this table is filled, and furthermore the ACPI does not make sense for MIPS processors.

Here we propose the concept of system topology matrix, the system topology matrix only needs to know how many cores are there in the system. Dirk proposed a similar concept of system distance matrix [5], but their matrix needs information from the system initialization and cannot be filled automatically.We implement the system topology matrix in the kernel after the kernel get the number of the cores in the system, for example, if the system has 8 cores, then we create a double dimensional array with ST_matrix [8][8] (see Fig. 21.5) to express the topology of the system. ST_matrix [0][1] means the distance between the core0 and core1, we normalized the data such that the cores on the same node results in a value of 1.

We use an average latency test to measure the communication latency between eight threads running on all eight cores, each test process is bounded on a core. For high performance technical computing application, the connect latency and the memory bandwidth frequently are the critical performance bottleneck, thus optimizing application code for connect latency and memory bandwidth is very important. Fig. 21.6 shows the results of our latency tests. The measured matrix

	Core0	Core1	Core2	Core3	Core4	Core5	Core6	Core7
Core0	0	1	1	1	1	1	1	1
Core1	1	0	1	1	1	1	1	1
Core2	1	1	0	1	1	1	1	1
Core3	1	1	1	0	1	1	1	1
Core4	1	1	1	1	0	1	1	1
Core5	1	1	1	1	1	0	1	1
Core6	1	1	1	1	1	1	0	1
Core7	1	1	1	1	1	1	1	0

Fig. 21.5 Initialization of the system topology matrix ST_matrix [8][8]

	Core0	Core1	Core2	Core3	Core4	Core5	Core6	Core7
Core0	0	1	1	1.5	4	5	4	6
Core1	1	0	1.5	1	4	5	5	6
Core2	1	1.5	0	1	5	6	4	5
Core3	1.5	1	1	0	5	6	4	5
Core4	4	4	5	5	0	1	1	1.5
Core5	5	5	6	6	1	0	1.5	1
Core6	4	5	4	4	1	1.5	0	1
Core7	6	6	5	5	1.5	1	1	0

Fig. 21.6 Normalized score of system topology matrix

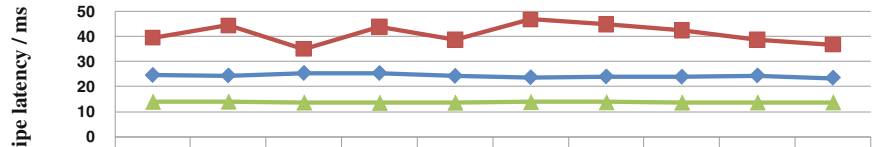
depict huge distance differences between remote nodes and we reset the cores to two nodes, core0 to core3 to node0 while core4 to core7 to node1.

After we get the system topology information according to the topology test module, we reset the cpuset of the system and ergodic the processes in the system once to bound them to the different nodes use the *schedule_set_affinity*(). Any process created after the test module will be automatically bound to a node. Then the CFS scheduler can schedule these processes on the node and remote memory is eliminated.

21.4 Evaluation

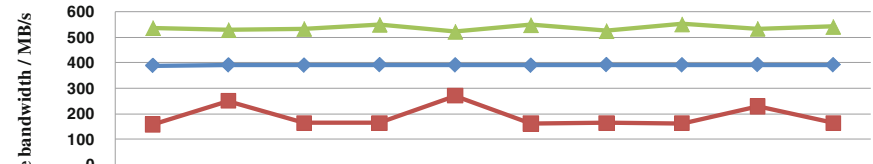
In this section we evaluate our architecture aware task-bound approach with the same environment we used in Sect. 21.2. We evaluate the benchmark with the task-bound approach on and off.

The task-bound approach can help CFS scheduler magically on the NUMA platform according to our tests. As depicted in Figs. 21.7 and 21.8, the test scores are far more excellent than the original CFS scheduler and are also better than the O(1) scheduler with NUMA patch. The pipe latency has been reduced by 66 % on average compared with the CFS scheduler, and also smaller than the average of the O(1) scheduler about 42.9 %. The pipe bandwidth has been upgraded by more than 196 % on average relative to the CFS scheduler and also 37 % higher than the O(1) scheduler on average. Besides, our test scores with task-bound approach are



	1	2	3	4	5	6	7	8	9	10
O(1)	24.48	24.29	25.29	25.27	24.11	23.58	23.89	23.78	24.31	23.39
CFS	39.52	44.42	35.07	43.87	38.67	46.81	44.77	42.43	38.73	36.78
CFS+TB	14.03	14.09	13.73	13.61	13.69	13.9	13.82	13.72	13.76	13.78

Fig. 21.7 Contrast test with lat_pipe in LMbench



	1	2	3	4	5	6	7	8	9	10
O(1)	389.27	391.28	390.25	392.18	391.23	389.78	392.47	391.39	392.59	391.69
CFS	158.4	251.38	164.13	164.3	271.47	161.89	164.39	163.75	230.21	164.18
CFS+TB	537	528.95	533.65	549.88	522.37	548.32	525.09	551.29	534.11	541.51

Fig. 21.8 Contrast test with bw_pipe in LMbench

very stable. Our evaluation demonstrates that our task-bound is significantly useful to help the CFS scheduler work efficiently on NUMA systems.

21.5 Related work

Research on NUMA related system optimizations dates back many years. Many research make efforts to address the computation and related memory on the same node [5, 6–8, 9]. None of the previous efforts, however, addressed automatic sort the system source and bound the task to the subsystem.

Li et al. [10] analyzed the O(1) scheduler and introduced a hierarchical scheduling algorithm based on NUMA topology. Their algorithm depends on the topology information provided by the system initialization and can not be used on the CFS scheduler anymore. Our algorithm is based on the architecture-aware module and can be ported to other platforms.

Blagodurov et al. [11] promoted the concept of resource conscious and presented a contention-aware scheduler, they identified threads complete for shared resources of a memory domain and placed them into different domain while put the independent processes on the same node, they tried to keep processes and their

memory on the same memory domain. Kamali in his master thesis [12] demonstrated the influence of remote memory access to the NUMA systems.

Dirk et al. [9, 13] proposed a platform-independent approach to describe the system topology, they use a distance matrix to provide system information, but their implantation depends on the user-defined strategies, only expert users can take advantage of their approach. Bosilca [5] proposed a framework as a middleware of MPI to tune types of shared memory communications according to the locality and topology.

21.6 Conclusion

Researchers have discovered that the original CFS scheduler fails to work efficiently on NUMA platforms due to lacking of consideration to the underlying hardware. Remote memory access occurring when the scheduler assigns tasks fairly on all the nodes. To address this problem, researchers devise the architecture aware task-bound approach. The evaluation shows that task-bound approach is of signality to the efficient work of CFS scheduler on NUMA systems. The research has a great significance to the development and popularity of domestic operating system.

Acknowledgments Our research is sponsored by the National "Core electronic devices high-end general purpose chips and fundamental software" project under Grant No.2010ZX01036-001-001, the Hi-tech Research and Development Program of China (863 Program) under Grant NO.2011 AA01A205, the National Natural Science Foundation of China under Grant NO.60973007, the Doctoral Fund of Ministry of Education of China under Grant NO.20101102110018, the Beijing Natural Science Foundation under Grant NO.4122042, the fund of the State Key Laboratory of Software Development Environment under Grant NO.SKLSDE-2012ZX-07.

References

1. Burkhardt, H.: KSR. June 2011 | TOP500 Supercomputing Sites (2011)
2. Jones, T.: Inside the Linux scheduler——The latest version© Copyright IBM Corporation (2006)
3. Jones, T.: Inside the Linux 2.6 Completely Fair Scheduler© Copyright IBM Corporation (2009)
4. Hewlett-Packard, Intel, Toshiba.: Advanced configuration and power interface (2011)
5. Ma, T., Bosilca, G., Bouteiller, A., Dongarra, J.J.: Locality and topology aware intra-node communication among multicore CPUs. In: Proceedings for EuroMPI, pp. 265–274 (2010)
6. Li, T., Baumberger D, et al.: Efficient operating system scheduling for performance-asymmetric multi-core architectures. In: Proceedings of Supercomputing (2007)
7. Azimi, T., et al.: Thread clustering: sharing-aware scheduling on multiprocessors. In: Proceedings of Eurosys (2007)
8. Corbalan, J., Martorell, X.: Evaluation of the memory page migration influence in the system performance. In: Proceedings of super computing, pp. 121–129 (2003)

9. Blagodurov, S., et al.: User-level scheduling on NUMA multicore systems under Linux. ACM Trans. Comput. Syst. **28**(4), Article 8 (2010)
10. Li, X.: NUMA scheduling algorithm based on affinitive node. Comput. Eng. 32(1), 99–101 (2006)
11. Blagodurov, S., Zhuravle, S., et al.: A case for NUMA-aware contention management on multicore systems. In: PaCT, pp. 557–558 (2010)
12. Kamali, A: Sharing Aware Scheduling on Multicore Systems. Simon Fraser University, Burnaby (2010)
13. Schmidl, D.: Towards NUMA support with distance information. In: IWOMP'11 Proceedings of the 7th International Conference on OpenMP, pp. 69–79. Springer, Berlin, Heidelberg © (2011)