Journal of Systems and Software
Manuscript Draft

Manuscript Number: JSS-D-17-00051

Title: Kernel-level Memory-Aware Load Balancing Mechanism on NUMA Multicore Systems

Article Type: Research Paper

Keywords: NUMA; scheduling; load balancing; remote memory access; resource contention

Abstract: Multicore systems with Non-Uniform Memory Access (NUMA) architecture are getting popular in computer systems. In NUMA systems, processor cores take longer time to access memories on remote NUMA nodes than on local nodes. To fully utilize memory resources, a process and its allocated memories may be distributed on different nodes. Remote memory access and contention for inter-node interconnect cause significant performance degradation. Performance gets better if the kernel scheduler is aware of hardware information and implements strategies to place processes' data on their local node or migrates pages appropriately to reduce remote memory access.
We propose a mechanism named kernel-level Memory-aware Load Balancing (kMLB), to reduce remote memory access and resource contention in NUMA systems. The Linux kernel is modified to keep track of each process's memory usage on each node. The load balancing procedure is modified to incorporate the proposed memory-aware policies. These policies choose suitable processes for inter-node load balancing according to processes' memory usages on all nodes, so that each process has a higher chance of running on the node along with most of its allocated memory. Experiment results demonstrate the proposed kMLB significantly improves system performance by effectively reducing remote memory access and inter-node resource contention.

Highlights

- We propose a mechanism named kernel-level Memory-aware Load Balancing (kMLB) to reduce remote memory access and resource contention in NUMA systems.
- The Linux kernel is modified to keep track of each process's memory usage on each node.
- The load balancing procedure is modified to incorporate the proposed memory-aware policies.
- The proposed memory-aware policies choose suitable processes for inter-node load balancing according to processes' memory usages on all nodes, so that each process has a higher chance of running on the node along with most of its allocated memory.
- Experiment results demonstrate the proposed kMLB significantly improves system performance by effectively reducing remote memory access and inter-node resource contention.

Kernel-level Memory-Aware Load Balancing Mechanism on NUMA Multicore Systems

Mei-Ling Chiang[1]   Shu-Wei Tu[2]

Department of Information Management,
National Chi Nan University,
Nantou, Puli, Taiwan, R.O.C.
Email: {joanna[1], s101213531[2]}@ncnu.edu.tw

Abstract

Multicore systems with Non-Uniform Memory Access (NUMA) architecture are getting popular in computer systems. In NUMA systems, processor cores take longer time to access memories on remote NUMA nodes than on local nodes. To fully utilize memory resources, a process and its allocated memories may be distributed on different nodes. Remote memory access and contention for inter-node interconnect cause significant performance degradation. Performance gets better if the kernel scheduler is aware of hardware information and implements strategies to place processes' data on their local node or migrates pages appropriately to reduce remote memory access.

We propose a mechanism named kernel-level Memory-aware Load Balancing (kMLB), to reduce remote memory access and resource contention in NUMA systems. The Linux kernel is modified to keep track of each process's memory usage on each node. The load balancing procedure is modified to incorporate the proposed memory-aware policies. These policies choose suitable processes for inter-node load balancing according to processes' memory usages on all nodes, so that each process has a higher chance of running on the node along with most of its allocated memory. Experiment results demonstrate the proposed kMLB significantly improves system performance by effectively reducing remote memory access and inter-node resource contention.

Keywords: NUMA, scheduling, load balancing, remote memory access, resource contention

1. INTRODUCTION

Multicore systems are popular in computer systems since they have parallel computing ability and are more economic in resource utilization compared to systems with single-core architecture. In a multicore processor, processor cores are able to access shared resources such as cache resources, memory, and/or input/output devices. When they demand the same resources at the same time, processor cores compete with each other for the shared resources. Symmetric Multi-Processing (SMP) [1] architecture is widely used. In this kind of architecture, each processor core can access any shared resource through the common bus. Though the system gains higher computing power by increasing the number of processor cores, more chances for processor cores to compete with each other occur. As the number of cores increases, resource contention is more severe and leads to performance degradation. Furthermore, the number of cores in SMP systems is limited.

Resource contention is a crucial problem in SMP systems. Non-Uniform Memory Access (NUMA) [2] platforms have become more and more popular because they have higher scalable hardware design than SMP systems. NUMA architecture divides resources like processor cores and the memory of the system as NUMA nodes. Processor cores access memory at the same node (called the local node) at a faster speed, whereas, they access memory at other nodes (called remote nodes) with the latency of remote access. Inter-node resource contention occurs when processor cores use the interconnect links to access remote memory. Previous research [3-14] indicates that the inter-node resource contention reduces system performance significantly in NUMA systems.

Several studies [3-8] have proposed scheduling techniques that arrange tasks with complementary characteristics to run on cores simultaneously for reducing resource contention. In this paper, a task refers to a schedulable entity such as a process or a thread for kernel scheduling. Researchers [3,4] have also found that when contention-aware schedulers designed for Uniform Memory Access (UMA) [15] architecture run on the NUMA architecture, they do not effectively decrease resource contention and sometimes even cause severe performance degradation. The main reason is a lack of consideration of the latency of accessing memory in remote nodes and the interconnect contention. For NUMA systems, reducing remote memory access can also reduce the resource contention caused by competition between processors for the inter-node connection.

Operating system schedulers ordinarily regard each processor core as an individual processor, even in NUMA systems. For example, in the Linux kernel, every processor core has a runqueue. When a task is ready to run, it is assigned to a runqueue by the kernel scheduler and waits for execution. To effectively utilize processor resources, tasks may be migrated to other runqueues. The kernel scheduler migrates tasks from a higher loaded processor core to a lower one. Therefore, when the kernel balances the loading of processor cores on the NUMA system, the kernel may migrate tasks from one node to another. Whereas, their allocated memories are not migrated at the same time ordinarily. After a task is migrated, it may still request memory allocation while it is running. Operating systems usually allocate memory in the local node where the task is currently running. Therefore, if a task has been migrated, its allocated memory may be scattered among different nodes. If a task accesses its previously allocated memory after being migrated from its local node, this causes latency of remote memory access and performance degradation.

A straightforward way to reduce remote memory access is to constrain a task and its memory to use resources on specific nodes. However, it is difficult for the kernel scheduler to balance CPUs' loading if all tasks are bound to run only on their current nodes. To alleviate remote memory access, some researches [9-11] proposed on-demand page migration, in which the kernel moves only the pages that a task actually accesses to the node where the migrated task is currently running. Similarly, the automatic page migration [12-14] mechanism is implemented in the Linux kernel. These mechanisms are completed by page table invalidation and page fault handling to migrate referenced pages to the current node where the task is running. Moreover, page fault handling is done in the context of the task that causes the fault, which increases additional processing overhead for memory access.

If inter-node task migration is required for balancing CPUs' load, it is important to select suitable tasks for migration, due to the high cost of memory migration along with page fault handling. Because tasks' memories are spread on multiple nodes, their chances of incurring remote memory access are different. However, we found that most researches do not consider the effect of task selection in the kernel's load-balancing activity. For example, in the default Linux kernel, the first movable task in the busiest CPU's runqueue is migrated in the load balancing mechanism. The Linux scheduler does not take a task's memory usage on each node into account since the kernel does not separately count the memory usage of each task on each node.

To reduce remote memory access and resource contention on multicore NUMA systems, we have proposed a kernel-level Memory-aware Load Balancing (kMLB) mechanism and implemented it in the Linux kernel. The load balancing mechanism is enhanced to incorporate a memory-aware task selection procedure for choosing suitable tasks for migration among nodes. In our implementation, the kernel is modified to record the memory usage of each task on each NUMA node. According to the memory usage information, the task expected to incur less remote memory access is selected for migration. We then modify the load balancer of the Linux kernel so that it migrates the proper tasks according to the proposed task selection policies. Experiment results show that the proposed kMLB mechanism effectively reduces remote memory access and resource contention among processor cores. As a result, system performance is increased as well.

2. BACKGROUND AND RELATED WORKS

This section introduces some background technologies and related works. Section 2.1 introduces multicore and multi-processor architecture. Section 2.2 describes the techniques and algorithms of task scheduling and memory policies used in the default Linux kernel. Section 2.3 presents some related works. Section 2.4 introduces Dynamic Task-aware Scheduling mechanism (DTAS) [8] that we employ to further reduce resource contention.

2.1 Multicore and Multi-processor Architecture

Modern computing systems mostly operate using multicore and/or multi-processor architecture. Main processor manufacturers have a variety of products such as Intel i5, Intel i7, and AMD Phenom X2 processors. They belong to the multicore processor family. The multicore and multi-processor architecture is a system that integrates several cores into a single processor, and the whole system contains multiple processors in it. A multicore processor usually runs with a lower frequency than a single-core processor, but multiple cores have the parallel computing ability.

Multicore and multi-processor system can be implemented in different ways. The most common one is SMP architecture [1], in which every CPU shares system resources like main memory and input/output devices. All CPUs use the same bus to access memory and have the same access time to access any memory address in the system. If two or more CPUs access memory simultaneously, CPUs compete with each other for the shared resources. As a result, the number of cores in the whole system is limited in SMP architecture.

NUMA architecture [2] solves the scalability problem of SMP architecture. Figure 1 shows an example of NUMA architecture, in which a NUMA system contains multiple nodes, and each node has resources like CPUs, memory, or I/O devices. Processors can access memory in the local node or in the remote node. Processors access memory regions in the remote node through inter-node interconnect links with extra access latency.

Most of NUMA systems use Cache Coherent NUMA architecture (ccNUMA) [2] which maintains cache coherence across shared memory. Because of cache coherence maintenance and resource contention, ccNUMA may perform poorly when multiple processors attempt to access the same memory area at the same time.

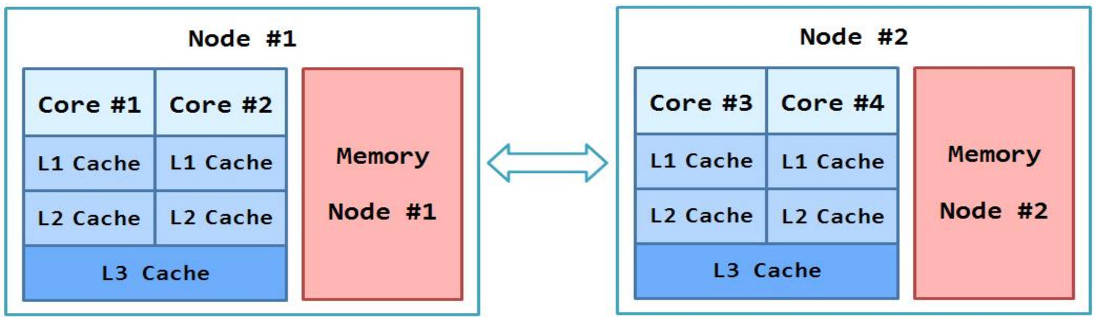


Fig. 1. An example of NUMA architecture [2].

2.2 Linux Scheduling and Memory Allocation Mechanisms

Section 2.2.1 introduces Linux's default scheduling mechanism for task assignment and the load balancing mechanism. Section 2.2.2 introduces Linux's memory policies that control memory allocation for each process on NUMA nodes. Section 2.2.3 introduces some NUMA-related system calls that support memory migration between nodes.

2.2.1 Linux Kernel Scheduler and Load Balancing Mechanism

In an operating system, the kernel scheduler is responsible for dispatching runnable tasks to run on CPUs. The execution order and execution time quantum of each task are determined by the kernel scheduling mechanism. The Linux kernel uses the Complete Fair Scheduler (CFS) algorithm [16], since Linux kernel version 2.6.23. The main concept of the CFS scheduler is total fairness; that is, the scheduler ensures that each task obtains its fair share of execution time and avoids tasks being starved.

In Linux, each processor core has a runqueue and every runnable task is assigned to a runqueue by the kernel scheduler to wait for execution. For load balancing, the least loaded CPU is usually selected for a task's execution. When the loading of processor cores is unbalanced, the kernel scheduler executes the load balancing procedure to push some tasks from the busiest CPU to the idlest CPU. When a processor core is idle and its runqueue is empty, it then pulls a task from the busiest CPU and executes the task. Therefore, a task may be migrated across different processor cores, even on different NUMA nodes.

In Linux, the hardware dependency of CPUs can be illustrated by a hierarchical scheduling domain tree [17]. The scheduler reduces the number of task migrations across nodes by way of the scheduling domains. A distinct scheduling domain is associated with every node on the system. With scheduling domains, the scheduler prefers migrations within a domain to inter-domain migrations.

When the scheduler balances the loading of CPUs, it starts from the most relevant CPUs in a smaller scheduling domain, and then balances the loading of CPUs in a larger scheduling domain. That is, it first finds a CPU with the least loading, and then searches the scheduling domain for a CPU with the highest loading. The scheduler then migrates one or more tasks from the busiest CPU to the idlest one. If all tasks in the busiest CPU's runqueue cannot be migrated to the idlest CPU, the scheduler then searches the next scheduling domain for the busiest CPU in the domain and then repeats the work until the scheduler migrates one or more tasks to the idlest CPU.

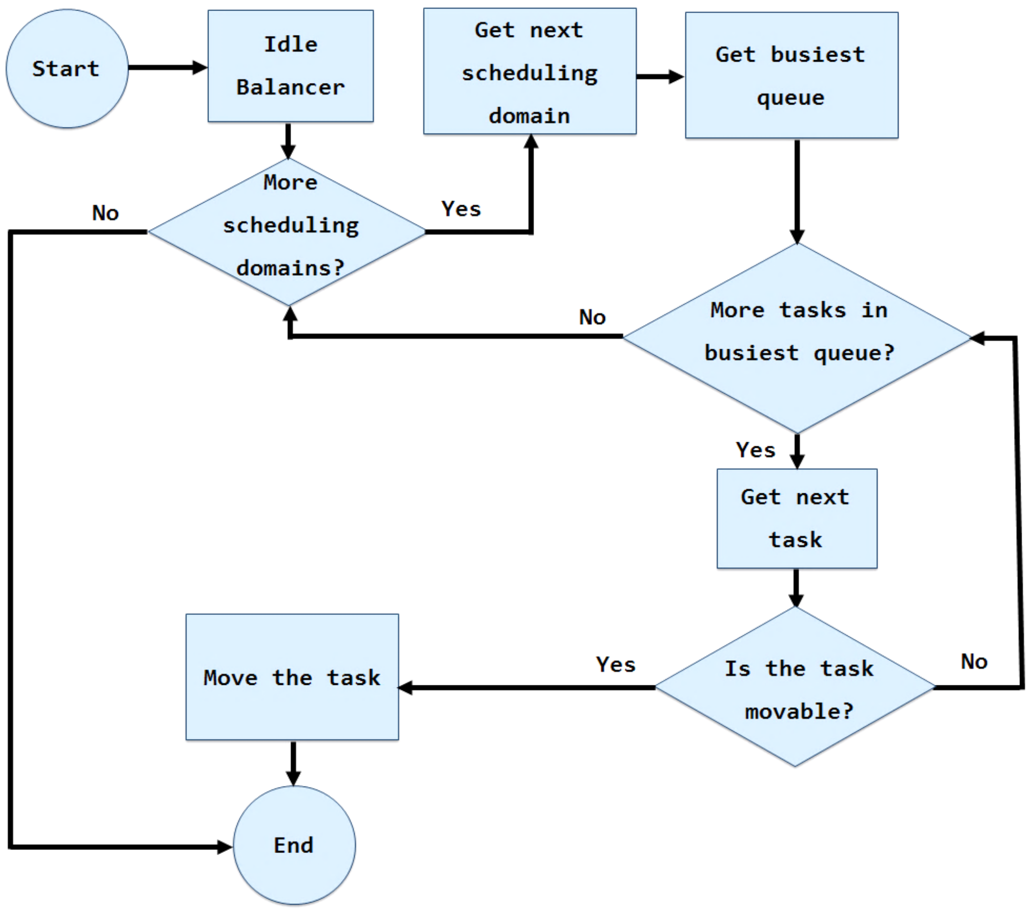


Fig. 2. The idle balancing procedure in the Linux kernel.

When a CPU is idle, the scheduler is also invoked to perform the idle balancing procedure. As shown in Figure 2, the kernel scheduler pulls tasks from the busiest CPU's runqueue to the idle CPU's runqueue. The kernel scheduler always migrates the first movable task in the runqueue if these two CPUs have imbalanced loads. If the chosen task is currently running, it is skipped and the idle balancer picks the next task.

2.2.2 Linux Kernel NUMA Memory Policies

There are different memory allocation methods for tasks in the Linux kernel on NUMA systems. The default memory allocation policy is used for all tasks unless users or programmers invoke the *set_mempolicy()* system call to change their policies, so that the kernel allocates memory only to the nodes specified in the *nodemask* variable. The *nodemask* of a task is a bitmap for specifying the nodes that its memory can be allocated to.

The Linux kernel provides four memory policies to control memory allocation for each task as follows:

- **Default**. The Linux kernel attempts to allocate memory for a task on the local memory node where it is currently running. If the current NUMA node is out of memory, the kernel will allocate memory pages on another NUMA node. For load balancing, sometimes a task is migrated from a CPU on the current node to one on another node, while its data remains on the previous node. As a result, the task accesses its previously allocated memory on the remote node, while accessing newly allocated memory space on the current node.

- **Interleave**. The Linux kernel uses the round-robin algorithm to allocate memory on every NUMA node in the specific node list which is defined by the kernel or programmers. Programmers can invoke system calls such as *mbind()* or *set_mempolicy()* to set the memory policy for a task to run on specific nodes.

- **Prefer node**. Memory allocation occurs on a specific NUMA node. The scheduler first attempts to allocate memory on the designated node, and then allocates memory on other nodes if there is not enough free memory on that node.

- **Bind.** The kernel allocates memory for a task only on the specific nodes which are set in the *nodemas*k. The kernel starts memory allocation on the first node in the *nodemask*, and then allocates memory on the second node and so on if there is no free memory on the previous node. The kernel does not allocate memory on nodes which are not specified in the *nodemask*.

2.2.3 NUMA-related System Calls

In a NUMA system, the memory is divided into multiple NUMA nodes. Accessing memory on the remote nodes through inter-node interconnect slows down the access speed. To offer better support for NUMA systems, Linux provides the CPUSET [18] mechanism for users to constrain which CPUs and NUMA nodes are used by a task or a set of tasks. The Linux kernel also implements several NUMA-related system calls, such as *get_mempolicy()*, *mbind()*, *migrate_pages()*, *move_pages()*, and *set_mempolicy()*. These system calls are provided for programmers to set the memory policy and control memory allocation for a task.

Users or programmers can invoke *set_mempolicy()* to change a task's memory policy, or invoke *get_mempolicy()* to obtain the memory policy of a task. Other system calls such as *mbind()*, *migrate_pages()*, and *move_pages()* also support memory migration on NUMA systems. The *mbind()* system call is used to set the memory policy for the memory region of a task. When the kernel allocates memory for the task, it starts to allocate memory on nodes specified in the *nodemask* argument and applies the specified memory policy. If some pages have already been allocated on nodes not in the *nodemask*, the kernel migrates them to the nodes in the node list. The *migrate_pages()* system call is used to migrate all pages of a task from one set of NUMA nodes to another. The *move_pages()* system call is used to migrate a specific set of memory pages of a task to a set of nodes. In addition, the user-level library named *libnuma* and the command-line utility named *numactl* also help users or programmers to issue NUMA-related system calls more efficiently.

2.3 Related Work

In NUMA systems, processors and system memory are divided into nodes. The memory in the same node with a processor is considered as local memory. Processor cores access local memory faster than they access remote memory. The Linux kernel always attempts to allocate local memory for a task. However, when the Linux scheduler balances CPUs' loading using task migration, tasks may be moved across nodes and access their previously allocated memory from a remote node. A straightforward way to reduce remote memory access is binding a task on the node to which its memory is allocated. However, this is detrimental to the balance of the CPUs' loading.

Linux provides the CPUSET [18] mechanism and *sched_setaffinity()* system call for users to bind tasks or memory on specific NUMA nodes. Programmers that have the knowledge of hardware and applications may also modify NUMA configurations in the proc file system [19] for better performance. Although Linux provides system calls for users to assist in reducing remote memory access, programmers can invoke related system calls such as *migrate_pages()* and *move_pages()* to migrate a task's memory pages to its currently running node. However, the cost of memory migration is expensive. To have proper memory migration, users require the allocation information of system memory to help them to make the migration decision.

Terboven et al. [9] implemented a *Next-touch* approach in Linux. By using the *mprotect()* system call to change protection on a memory region and a segmentation fault signal handler that invokes the *move_pages()* system call to migrate pages, the accessed page is migrated to the task's currently running node. Similarly, Goglin and Furmento [10] implemented a *Next-touch* approach in Linux. They presented two different implementations. The user-space implementation of the *Next-touch* approach is also done through the *mprotect()* system call and a signal handler for processing the segmentation fault. Kernel-based implementation is done the using *madvise()* system call and a modified kernel page fault handler. The results of the experiment showed that their kernel-based implementation is 30% faster than the user-space model and has a much lower base overhead when migrating small memory. The user-space implementation seems to be more suitable for migrating large memory buffers while keeping the knowledge of the migrated page locations.

Mishra and Mehta [11] also presented a memory migration on-demand policy that is similar to the *Next-touch* approach. It enables automatic dynamic migration of pages when they are accessed by a migrated task. The implementation is also done using the *madvise()* system call invoked by applications and a modified kernel page fault handler. In the recent version of the Linux kernel, the automatic page migration [12-14] mechanism has been implemented. In the kernel, after a task is migrated to another node, its pages are set to be invalid in the page table at certain intervals of time. The page fault handler is invoked when the task accesses the invalidated pages, and then those pages are migrated to the node the task is currently running on. Page fault handling is done in the context of the task that causes the fault, which increases additional overhead for memory access.

Zhuravlev et al. [7] surveyed many studies of scheduling techniques that address shared resource contention in multicore processors. Blagodurov et al. [3] indicated many contention-aware scheduling approaches designed for UMA systems work with lower efficiency on NUMA systems because of the remote memory access latency and the interconnect contention. They designed a user-level contention-aware scheduling algorithm named DINO for NUMA systems. DINO dynamically classifies tasks into three classes according to their memory access behaviors. It then arranges complementary tasks to run on cores in each NUMA node with a specific placement layout. When a task's class changes, it may be assigned to run on a different node. Therefore, the decision to migrate a task is based on shared resource contention instead of load balancing. DINO also detects and reduces superfluous task migration between nodes. In addition, they designed several memory migration strategies that determine which memory pages of a task to migrate to when the task is moved to a new node. A user-level daemon is implemented by invoking NUMA related system calls, and it is woken up periodically to perform this memory migration.

Merkel and Bellosa [5] investigated the effects of resource contention and indicated that if running tasks on different processors require the same resource, it not only slows down tasks' executions but also increases power consumption. They proposed a memory-aware scheduling mechanism based on runqueue sorting and frequency heuristic policies. The scheduling mechanism sorts tasks by their memory intensity, which is determined by the number of memory transactions obtained from the hardware performance counters [20]. If a task's memory intensity exceeds a predefined threshold, it is classified as memory-bound; otherwise it is compute-bound. Tasks in the runqueues of even-numbered processors are sorted according to descending memory intensity, whereas, they are sorted in ascending order for odd-numbered processors. The runqueue sorting policy mainly co-schedules memory-bound with compute-bound tasks to run on processor cores at the same time to reduce memory contention among processor cores. The frequency heuristic policy which properly switches processor frequency and frequency scaling is used only if contention cannot be avoided. Their experiments showed that it is better to run tasks with complementary characteristics simultaneously on cores that share memory resources to reduce resource contention among processor cores.

Merkel et al. [6] further enhanced their work [5] and employed the concept of task activity vectors for characterizing applications through resource utilization. An activity vector represents the degree to which a running task utilizes various processor-related resources. They then proposed vector balancing policy to reduce resource contention by means of task migrations guided by activity vector information. The purpose of this policy is to ensure that tasks with various resource utilization characteristics are available on each processor core for co-scheduling.

Knauerhase et al. [21] modified an operating system kernel to gather the runtime resource usage of each task from the performance monitoring counters. Based on the observation data, they then developed various scheduling policies to improve system performance in multicore systems. For example, to reduce the cache interference caused by processors' contention with each other in accessing cache memory, one policy aims to balance the cache load of cores and then distributes cache-heavy threads throughout the system. Furthermore, the observation data also affect task migration decisions in the load balancer. A task is overweight if its cache usage is greater than the average usage. The load balancer prevents migration of overweight tasks, both to avoid excessive migration costs and to prevent the creation of new cache interference in a new location. Their experiments demonstrated that the proposed observation-based scheduling has improved the system performance in multicore systems.

Dashti et al. [22] indicated that the memory traffic from data-intensive applications causes congestion on memory controllers and interconnect links, which degrades performance greatly on modern NUMA hardware. They then developed a memory placement algorithm named Carrefour in Linux. This algorithm composes the mechanisms of page co-location, interleaving, replication, and thread clustering. It improves performance by managing traffic on memory controllers and interconnections in NUMA systems. Srikanthan et al. [23] explored the issues of performance degradation due to resource contention and communication caused by sharing resources or accessing shared data. They developed a Sharing-Aware Mapper (SAM) for task to processor core assignments to separate traffic caused by data sharing from that occurring due to memory access. By using hardware performance counters to obtain tasks' runtime resource usage, it identifies tasks that share data and those that have high memory and/or cache demand. It then co-locates tasks that share data on cores and distributes tasks with high demand for cache and memory bandwidth across cores. In addition to addressing the issues of thread and data placement for NUMA systems, Lepers et al. [24] considered not only the placement of threads and data on the same or different nodes, but also the asymmetry of interconnect links. They then developed a new thread and memory placement algorithm named AsymSched to maximize the bandwidth for communicating threads.

2.4 The Dynamic Task-Aware Scheduling Mechanism

In our previous study [8], the Dynamic Task-Aware Scheduling mechanism (DTAS) was proposed to reduce resource contention in NUMA systems. DTAS identifies tasks as compute-bound or memory-bound according to their runtime memory access behavior. DTAS also assigns each processor core to compute-bound or memory-bound types. A core assigned to the compute-bound type executes only compute-bound tasks, while a memory-bound type core executes only memory-bound tasks, as illustrated in Figure 3.

Processor cores that share resources are arranged to run complementary tasks, in order to avoid resource contention. Moreover, because a task may have different resource requirements during its lifetime, it may be compute-bound at first and later become memory-bound. Therefore, the kernel scheduler monitors its resource usage and reclassifies its type dynamically during run time. The scheduler then assigns the task to the same type of processors for execution.

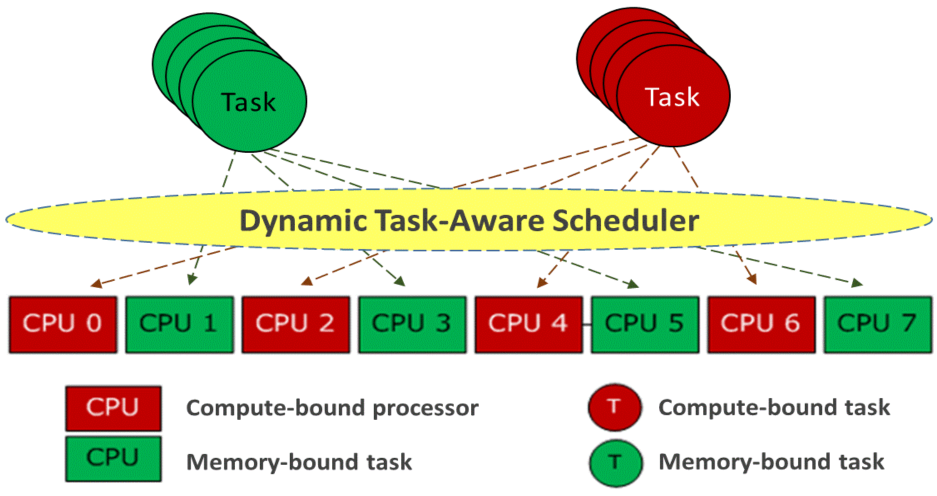


Fig. 3. The Dynamic Task-aware Scheduler dispatches tasks to processors according to their types.

If the amount of compute-bound and memory-bound tasks in the system is not balanced, the loading of CPUs is imbalanced. For example, if there are more compute-bound tasks, the loading of compute-bound CPUs will be higher than the loading of memory-bound CPUs. To avoid running an imbalanced amount of different types of tasks, causing some processor cores to stay idle, a kernel thread is created to monitor the loading of each CPU. It then dynamically adjusts the processor type for processor cores according to the ratio of compute-bound tasks to memory-bound tasks currently running in the system.

3 KERNEL-LEVEL MEMORY-AWARE LOAD BALANCING MECHANISM IN NUMA SYSTEMS

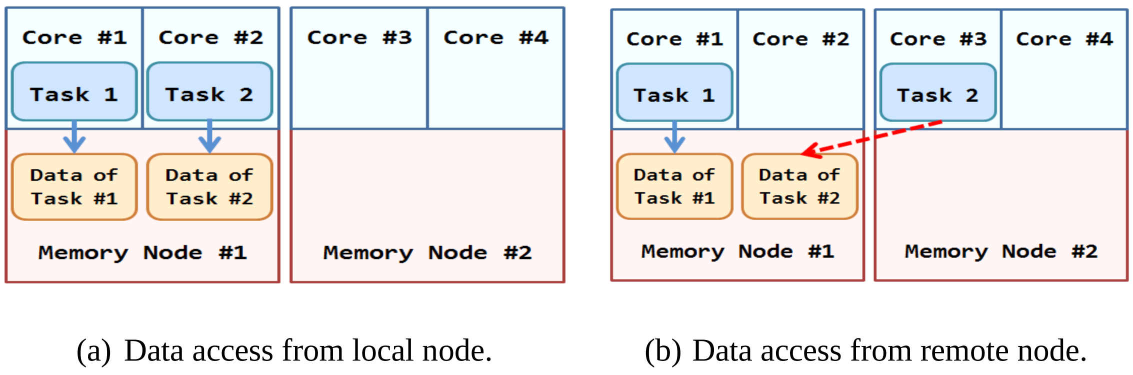
To reduce inter-node memory access in NUMA systems, we propose a kernel-level Memory-aware Load Balancing (kMLB) mechanism which changes the handling of inter-node task migration in the load balancing procedure. This section introduces our motivation and discusses the side effects of task migration in load balancing. The idea, design, and implementation of the proposed kMLB mechanism are then presented.

3.1 The Side Effects of Task Migration in Load Balancing

In a multicore and multi-processor system, every runnable task is assigned to the runqueue of a CPU and waits for execution. The execution time that each task needs is different, so the completion times of tasks are also different. To avoid circumstances where some CPUs are idle whereas others are busy, the kernel scheduler has to balance the loading of CPUs to make the best of CPU resources.

In order to use the CPU resources effectively, conventional operating systems such as the standard Linux kernel migrate tasks among processor cores to balance the load. Therefore, tasks may be migrated among NUMA nodes during their run time. However, task migration for load balancing may cause negative effects on system performance as follows.

Firstly, the increased remote memory access leads to performance degradation. Although a task is migrated, its allocated memories are not ordinarily migrated. Remote memory access thus occurs when a migrated task accesses previously allocated memory in the old node, as illustrated in Figure 4. If tasks in the system access remote memory, they cause contention among the interconnect links and remote memory controllers, too.

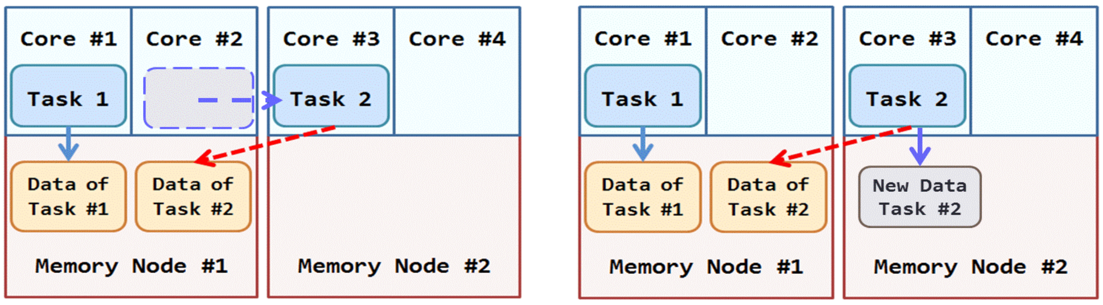


(a) Data access from local node. (b) Data access from remote node.

Fig. 4. Remote memory access caused by task migration.

Secondly, a process's allocated memories may be scattered to multiple nodes. This is because conventional operating systems usually allocate memory in the local node on which the task is currently running. Therefore, if a task has been migrated, its allocated memories may be distributed among different nodes, as illustrated in Figure 5. This increases the chance of remote memory access.

Thirdly, even if a task is migrated along with its allocated memories, it needs additional overhead for memory access. In order to reduce remote memory access, a straightforward method is to migrate a task along with all of its allocated memories from its original node to the target node. However, this migration incurs large cost and some pages may be not referenced at all after being migrated. This leads to useless page migration. To reduce the migration cost, some studies [9-11] have proposed on-demand page migration which migrates only the referenced pages. By way of marking the memory pages of the migrated task in the original node as invalid, the page fault handling procedure is triggered to migrate the faulted page once it is referenced.

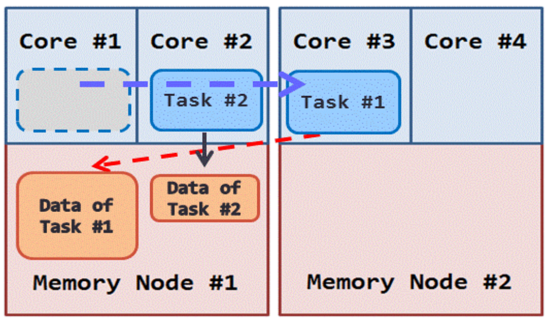


(a) A task is migrated. (b) Allocated memories are scattered.

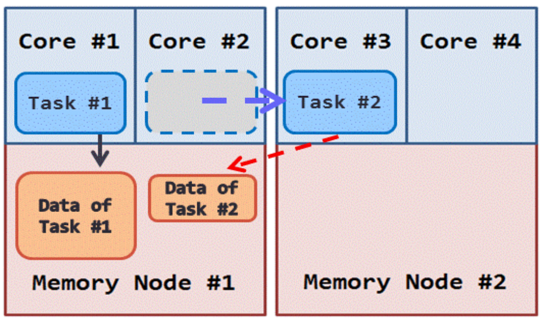
Fig. 5. Allocated memories are scattered due to task migration.

Similarly, the Linux kernel has implemented automatic page migration [12-14], in which the kernel moves only the pages that a task actually accesses to the node where the task is currently running. As a consequence, the allocated memories of a task are distributed between the nodes. This mechanism is also completed by page table invalidation and page fault handling to migrate accessed pages to the current node which the task is running on. This kind of page migration, though remote memory access is alleviated, must pay for the additional page fault handling cost.

Since migrating tasks between nodes for load balancing is inevitable, selecting suitable tasks for migration is thus very important. Because a task's allocated memories are scattered to several nodes, the cost of memory migration or page migration is dependent on the memory usage of each task. Some tasks have large memory requirements, while some do not. The amounts of memories allocated on the NUMA nodes are also different. Therefore, it is important to select the most suitable task for migration. For example, the task with the least possibility of accessing remote memory is a candidate. However, the Linux kernel scheduler balances the loading of CPUs by migrating tasks from a highly loaded CPU to the idlest CPU in the system. It uses the first-fit method to pick tasks from the busiest CPU's runqueue and does not consider their memory usage on each node. As shown in Figure 6, tasks with different memory usage incur different degrees of remote memory access when migrated.



(a) The larger task is migrated.



(b) The smaller task is migrated.

Fig. 6. Migrating tasks with different memory usage.

If task migration across nodes is needed for achieving better load balancing, it is better to select suitable tasks that incur less remote memory access. If a task's memory usage information on each node is obtained, the kernel scheduler can avoid migrating a task from a node that provides most of its memory space to other nodes. This can reduce the chances of future remote memory access after migration.

In this paper, we modify the Linux kernel scheduler to migrate suitable tasks between nodes according to their memory usage when it performs load balancing. In other words, the kernel scheduler avoids migrating a task with a higher chance of accessing the memory area on remote modes after being migrated to other nodes.

3.2 Design and Implementation

Generally, the kernel scheduler is invoked when a task finishes its time quantum or is preempted by other tasks. The scheduler picks another ready-to-run task in the current runqueue of the CPU and arranges the task for execution. If there is no task in the current CPU's runqueue, the CPU is idle. Therefore, the scheduler is invoked to execute idle balancing procedure which first attempts to find a task in another CPU's runqueue and then pulls it to the current runqueue. In the Linux kernel, the first-fit policy is used. That is, the first movable task that is allowed to run on the idle CPU is pulled and migrated.

For reducing remote memory access, we improve the default Linux kernel scheduling mechanism with the proposed kernel-level Memory-aware Load Balancing (kMLB) mechanism. Because the cost of inter-node page migration is high, we enhance the Linux kernel scheduler to select suitable tasks for migration in the inter-node load balancing procedure. The kernel scheduler is modified to take the memory usage of each task into account when selecting tasks for migration.

We have also designed and implemented several task selection policies. Basically, these policies use the memory usage information of each task on each node to select a suitable task for migration. The purpose is to avoid migrating a task from a node that provides most of its memory space.

Our implementation mainly includes three work in the Linux kernel. The first one is to design and implement task selection policies which select the most suitable tasks for migration among nodes. The aim is mainly to reduce remote memory access and page migration frequency. The second is to obtain memory allocation information of each task; this information is used in the task selection policy to choose tasks. We modify the Linux kernel to separately count the memory usage of each task on each node. The kernel scheduler then is modified to distinguish which task has a higher chance of accessing memory pages on remote nodes. The third is to modify the default load-balancing mechanism to incorporate the proposed task selection policies and use each task's memory usage on each node when selecting a task for migration.

3.2.1 Enhancing the Load Balancing Mechanism

We modify the Linux kernel's load balancing mechanism in order to offer better support for NUMA systems. Firstly, we modify the idle balancing procedure to incorporate the task selection policies. When a CPU is idle, the kernel scheduler is invoked and it tries to find the busiest CPU in the most relevant scheduling domain. It then selects a task that can be migrated to the idlest CPU. If none of the tasks in the busiest CPU's runqueue can be moved to the idlest CPU, the search procedure then falls back to the next highest relevant scheduling domain that includes more CPUs, and chooses a task that can be migrated.

Secondly, we design some policies in the idle balancing procedure which is activated in the inter-node load balancing procedure. The procedure uses the proposed policies to search the entire runqueue of the busiest CPU for a target task, instead of using the default first-fit method. As shown in Figure 7, in the modified idle balancing procedure, the scheduler chooses a target task according to the task selection policies from all tasks in the busiest CPU's runqueue and moves the task to the idle CPU's runqueue, instead of just checking the CPU's loading.

Besides the idle balancing procedure invoked by the idle processor core, the load balancing procedure is also executed in a certain time period. The CPU loadings are rebalanced and one or more tasks are migrated from the busiest CPU to the idlest CPU. Tasks may also be migrated to other nodes. In Linux kernel, the same routine is invoked in the load balancing procedure for moving tasks between nodes. An additional requirement can be set in finding the tasks that can be migrated in the load balancing procedure. For example, if a task has a relatively large amount of page frames on a node, it is not suitable to be migrated to another node.

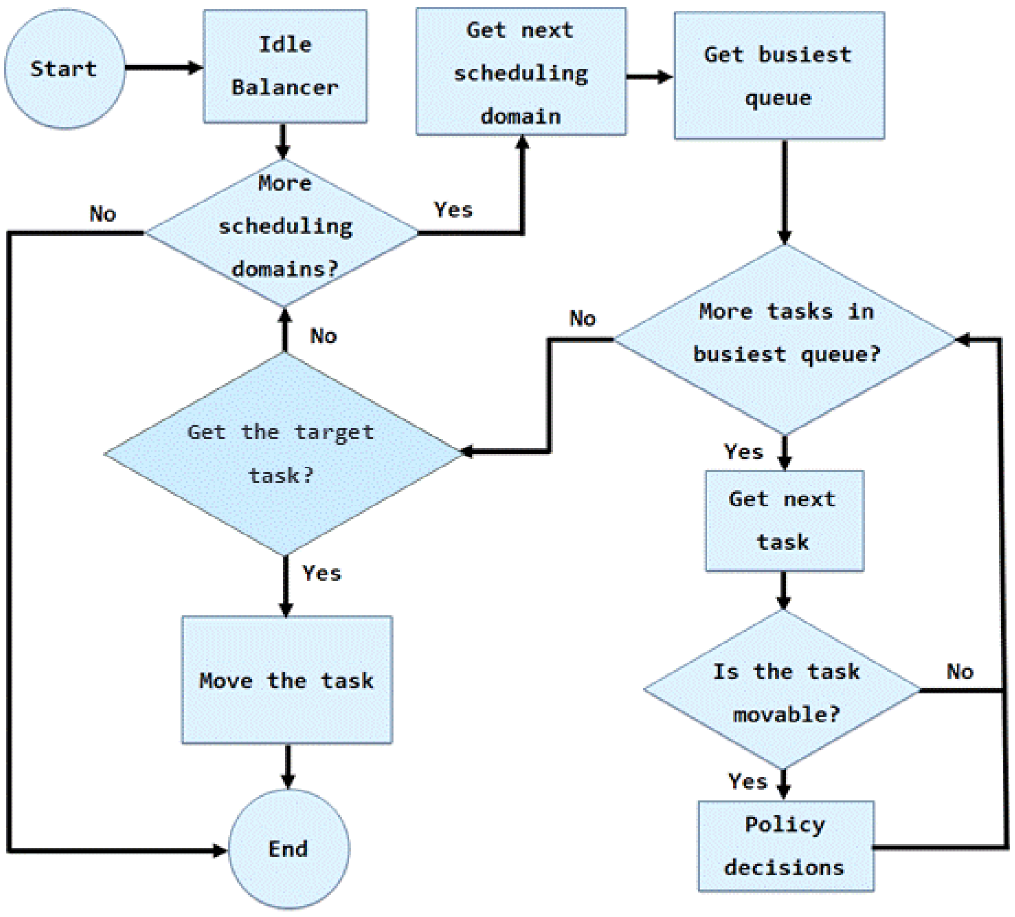


Fig. 7. The modified idle balancing procedure for incorporating policy decisions.

3.2.2 *Obtaining Tasks' Memory Allocation Information on Nodes*

In order to avoid migrating a task from its intensive memory usage node to others, the kernel scheduler needs more complete memory allocation information. Moreover, if a task is migrated to a new node, its new memory is allocated on the new node. Therefore, a task's allocated memory may scatter on several nodes. The kernel scheduler should migrate only the tasks having a higher chance of accessing remote memory.

The resident set size (RSS) [25] is the portion of memory occupied by a process that is held in main memory. In the Linux kernel, when the kernel allocates one or more page frames to a task, its RSS value is increased. While the kernel frees one or more page frames of a task, its RSS value is decreased. The RSS counter represents all of the memory page frames that a task is currently using.

For NUMA systems, we modify the data structure and related functions of the RSS counter in the Linux kernel to individually count the memory usage of a task on each node. That is, when the modified kernel allocates one or more pages to a task, the task's RSS value for the corresponding node is increased. While the modified kernel frees one or more pages of a task, its RSS value of the corresponding node is decreased. Table 1 shows the data structure of the RSS counter for each task. The original counter records only the total amount of memory pages, whereas it is split to sub-counters for different nodes.

Table 1. The *mm_rss_stat* structure for two NUMA nodes.

Original RSS Counter in 3.11.0	Modified RSS Counter	Description
MM_FILEPAGES	MM_FILEPAGES_0	File pages at node 0
	MM_FILEPAGES_1	File pages at node 1
MM_ANONPAGES	MM_ANONPAGES_0	Anonymous pages at node 0
	MM_ANONPAGES_1	Anonymous pages at node 1

Because copy-on-write mechanism is implemented in the Linux kernel, if a task is forked by its parent, unlike the parent, the task's RSS counter is reset to 0. The page frame is only actually allocated when the task is modifying write-protected pages or incurring page fault events. Therefore, the Linux kernel is modified to calculate each task's individual RSS counters in the exception handling routines such as the *do_wp_pages()* function and *do_fault()* function. As shown in Figures 8 and 9, we distinguish anonymous pages from file pages on each node for each task. In these examples, the system has two NUMA nodes.

Since the modified RSS counter represents the memory page frames that a task currently uses on each node, the kernel scheduler can use this counter to identify which task has a higher chance of accessing memory on remote nodes. Our task selection policies then make use of it to determine which one is the most suitable task for the inter-node task migration. Being aware of each task's memory usage on each node, the kernel scheduler is able to balance the inter-node CPUs' loading without misplacing tasks with their occupied memory areas.

```
if (page_to_nid(page))
   inc_mm_counter_fast(mm, MM_FILEPAGES_1);
else
   inc_mm_counter_fast(mm, MM_FILEPAGES_0);
```

Fig. 8. An example of separately counting the RSS counter for different nodes.

```
if (PageAnon(page)) {
   if (page_to_nid(page))
      rss[MM_ANONPAGES_1]++;
   else
      rss[MM_ANONPAGES_0]++;
} else {
   if (page_to_nid(page))
      rss[MM_FILEPAGES_1]++;
   else
      rss[MM_FILEPAGES_0]++;
}
```

Fig. 9. An example of separately counting the anonymous and file pages in the RSS counter.

3.2.3 Task Selection Policies for Migrating Tasks

The modified kernel scheduler chooses the task for migration according to the proposed task selection policies. Figure 10 illustrates an example of inter-node task migration. The idle CPU #5 invokes the kernel's idle balancing procedure which attempts to pull a task from another runqueue. The idle balancing procedure starts searching at the most relevant scheduling domain and ends with the biggest scheduling domain including all CPUs.

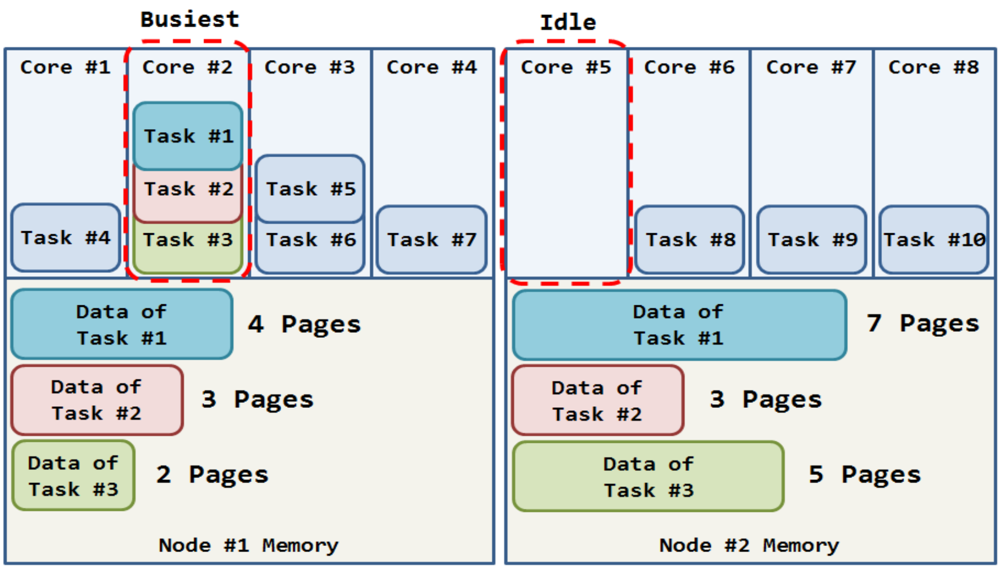


Fig. 10. An example of inter-node task selection and migration.

. Several task selection policies for inter-node migration are proposed as follows:

- **Default (i.e. First Fit).** This is the default policy used in the Linux load balancing mechanism. It examines tasks in the busiest CPU's runqueue in order and check whether tasks can be migrated. The default policy picks the first-fit task from the busiest CPU's runqueue for migration to the idlest one. Figure 11 shows the default policy.

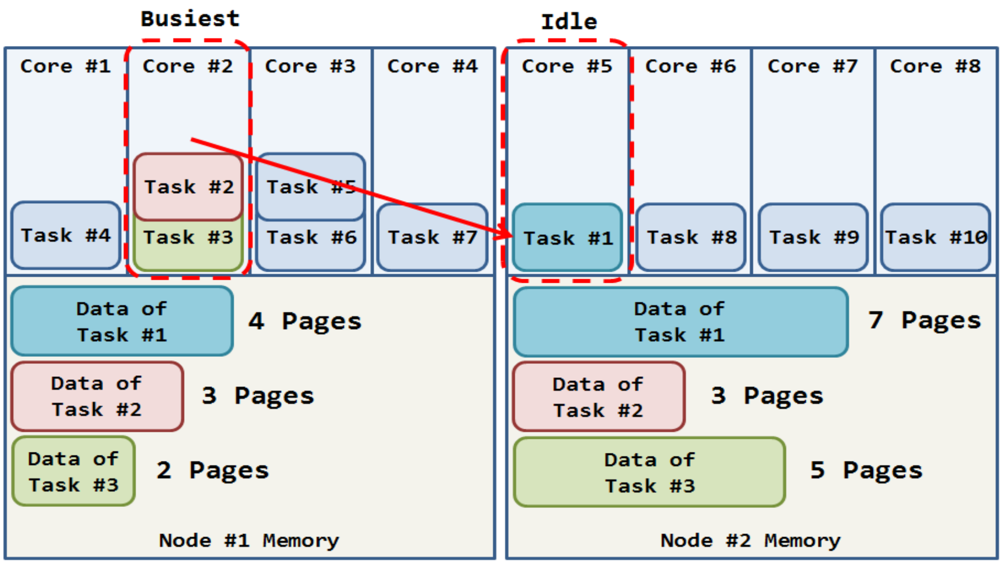


Fig. 11. The default first-fit policy.

- **Remote Min.** The Remote Min policy finds the task in the busiest CPU's runqueue with the smallest remote RSS value for migration to the idlest CPU. Here the remote RSS value of a task indicates the amount of its allocated page fames on the node where the busiest CPU resides. This policy is devised because the task after migration has the smallest chance of accessing memory on the remote node. If both the busiest CPU and the idlest CPU are on the same node, the Remote Min policy falls back to the default policy.

  As illustrated in Figure 12, this policy attempts to migrate a task (i.e. task #3 in this example) to the idlest CPU's runqueue. This task has the smallest number of page frames (i.e. 2 pages) allocated on the remote node (i.e. node #1) and has the smallest chance of accessing pages on the remote node.

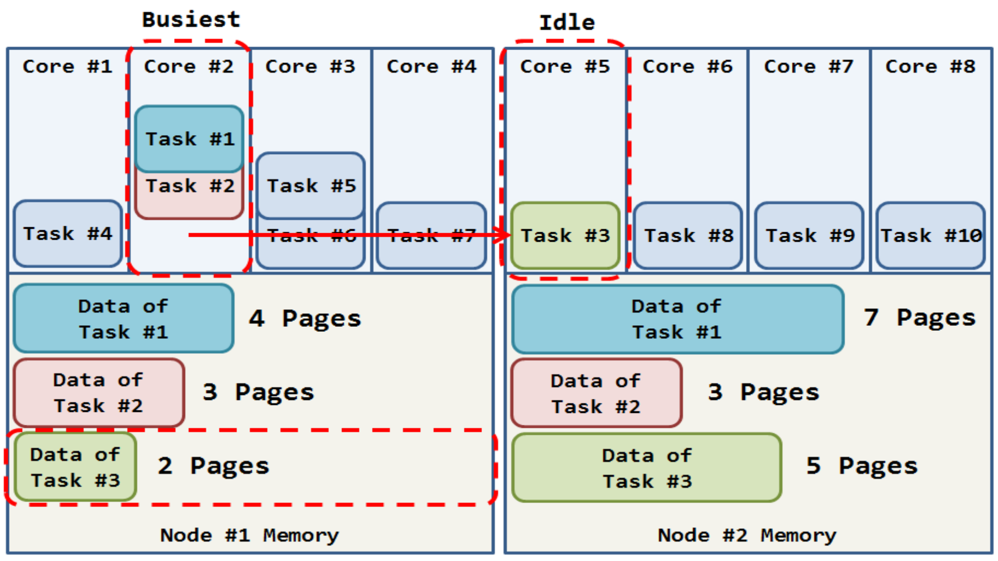


Fig. 12. The Remote Min policy.

- **Local Max.** The Local Max policy finds the task in the busiest CPU’s runqueue with the largest local RSS value for migration to the idlest CPU. Here the local RSS value of a task indicates the amount of its allocated page frames on the node where the idle CPU resides. This policy is devised because the selected task is the one with most remote memory access. In other words, more memory requests of the task after migration can be satisfied on the local node. If both the busiest CPU and the idlest CPU are on the same node, the Local Max policy falls back to the default policy.

  As illustrated in Figure 13, this policy attempts to migrate a task (i.e. task #1 in this example) to the idlest CPU’s runqueue. This task has the largest number of page frames (i.e. 7 pages) allocated on the target node (i.e. node #2).

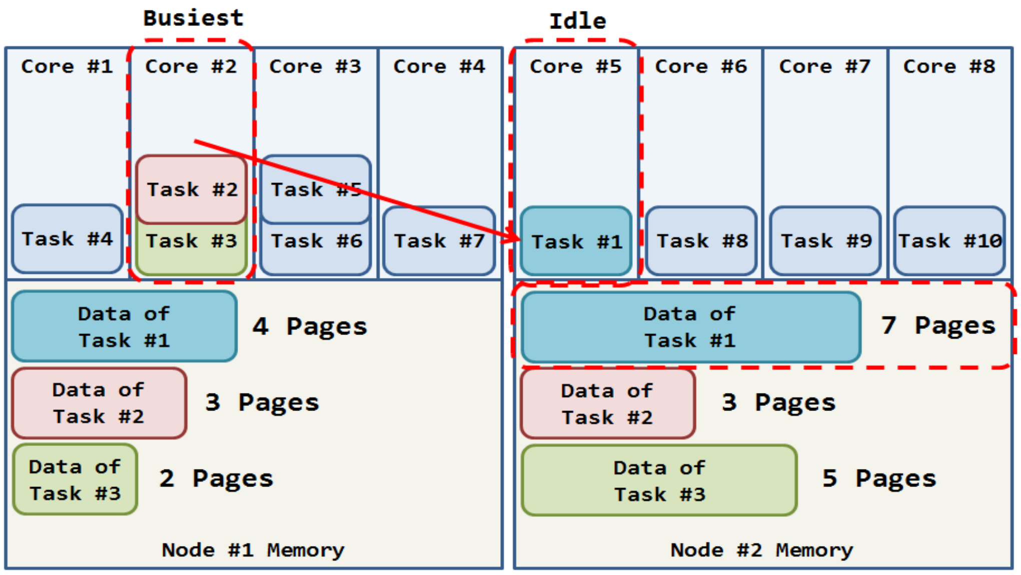


Fig. 13. The Local Max policy.

- **Total Min.** The Total Min policy finds the task in the busiest CPU's runqueue with the smallest total RSS value for migration to the idlest CPU. Here the total RSS value of a task indicates the total amount of its allocated page fames on all nodes. That is, the task to be migrated uses the least amount of page frames in the system. This policy is devised because it is expected to have least influence caused by task migration. If both the busiest CPU and the idlest CPU are on same node, the Total Min policy falls back to the default policy.

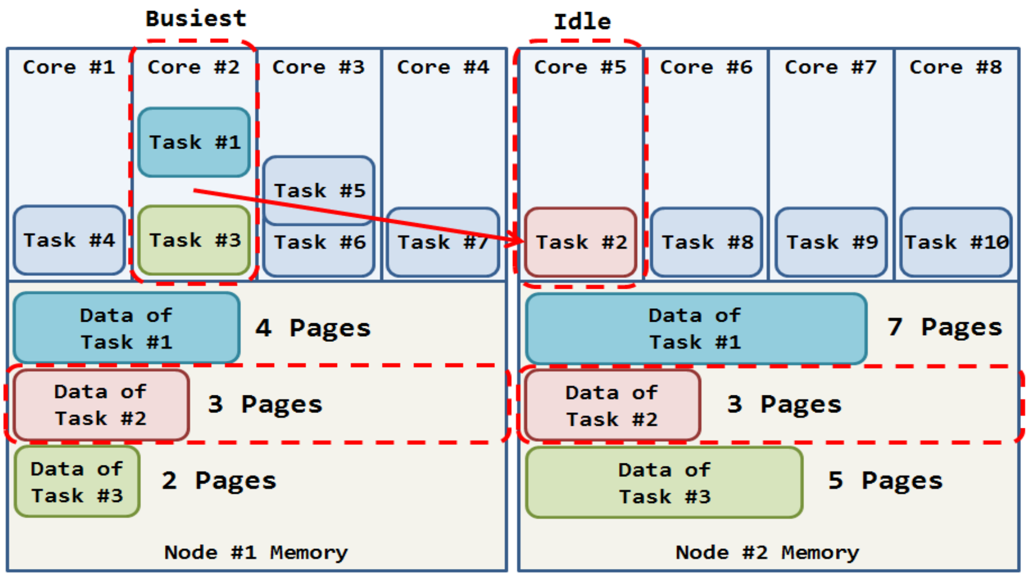


Fig. 14. The Total Min policy.

As illustrated in Figure 14, this policy attempts to migrate a smallest task (i.e. task #2 in this example) to the idlest runqueue. This task has the least amount of page frames (i.e. 6 pages) allocated on all nodes. The migrated task occupies the least amount of memory in the system.

4 PERFORMANCE EVALUATION

This section presents a performance evaluation of our kernel-level Memory-aware Load Balancing (kMLB) with various task selection policies. The performance improvement of the modified Linux kernel with the proposed kMLB mechanism over the default Linux kernel with the default kernel scheduler is measured. In particular, we also measure the performance improvement when the kMLB mechanism is incorporated into the system with the modified Linux kernel and the DTAS mechanism [8]. As introduced in Section 2.4, DTAS improves the system performance by scheduling complementary tasks to run on cores to avoid resource contention.

Section 4.1 describes our experimental environment. Section 4.2 introduces our experimental cases and the benchmarks used in the experiments. Section 4.3 presents the experiment results.

4.1 Experimental Environment

The NUMA system that we use in the experiments is an ASUS TS500-E6 server [26]. The server has two sockets and is installed with two Intel Xeon E5620 processors and 24 GB main memory. Its system layout [27] is shown in Figure 15. In this system, the NUMA factor, i.e. the ratio between remote memory access latency and local memory access latency, is 2.

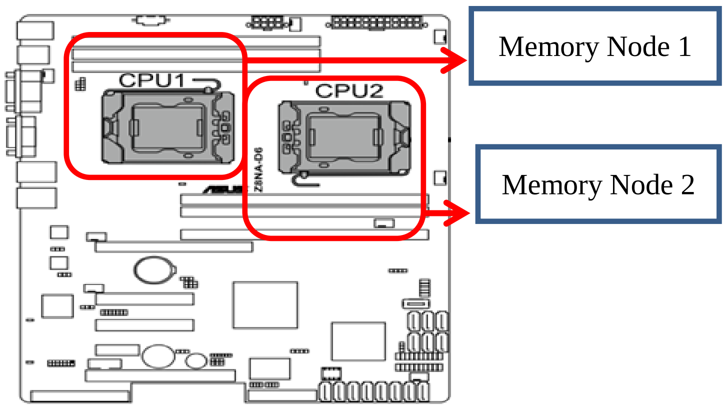


Fig. 15. The layout of the Z8NA-D6 motherboard in an ASUS TS-500 server [27].

The Xeon E5620 is a quad-core processor and supports Hyper-threading technology [28]. Because there are eight logical processors for a quad-core processor under Hyper-threading technology, so there are a total of 16 available logical processors in the system. The cores in one physical package share the L3 cache, and two logical processors in one physical core share the L1 and L2 caches, and execution units, as shown in Figure 16. The software and hardware specifications are shown in Table 2.

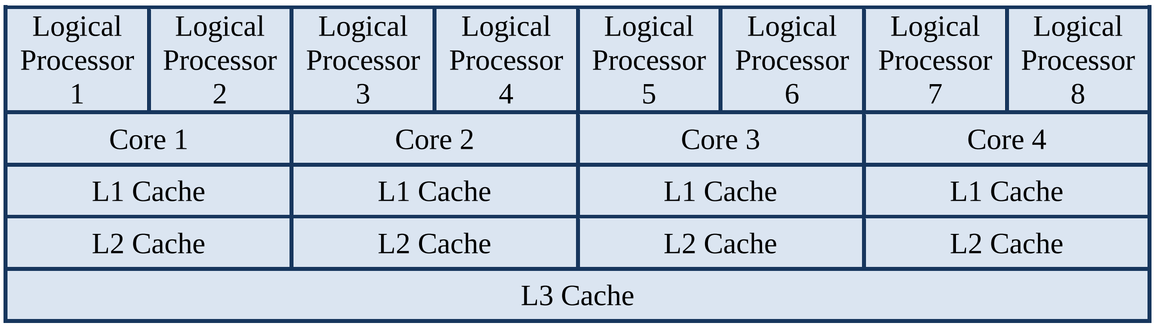


Fig. 16. The architecture of a Xeon E5620 processor.

Table 2. Software and hardware specifications.

Computer	**Asus TS500 Server**
Processor	Intel Xeon E5620 2.4GHz * 2
Memory	DDR3 2G * 4, DDR3 8G*2
Kernel Version	Linux 3.11.0
Benchmark	SPEC CPU2006 [29]
Motherboard	Z8NA-D6 [27]

4.2 Experimental setting

In order to evaluate the performance of the proposed kMLB mechanism, we design three experimental test cases that contain different task mixes to evaluate system performance under different mechanisms. The SPEC CPU2006 [29] benchmark suite that has 29 benchmarks for stressing the processor, memory, and compiler is used in the experiments. We execute 56 benchmarks simultaneously on our experimental system. In order to reach this number, some benchmarks run multiple instances. The benchmarks are shown in Table 3, the alphabetical letter in parentheses is the type of benchmark, “C” stands for a compute-bound task, and “M” stands for a memory-bound task.

These three experimental cases are designed as follows. Each node may have an imbalanced loading so that the kernel scheduler has to migrate tasks between nodes for load balancing. Test case 1 is the memory-bound case, in which the system runs more memory-bound tasks and thus suffers from severe resource contention. The system runs 35 memory-bound tasks and 21 computing-bound tasks. Test case 2 is the half-half case, in which the system runs 28 memory-bound tasks and 28 compute-bound tasks at the same time. Test case 3 is the compute-bound case which features less memory contention and cache contention. The system runs 21 memory-bound tasks and 35 compute-bound tasks. We run each test case 3 times and calculate the average elapsed time of each benchmark.

Table 3. Experimental cases (M: memory-bound, C: compute-bound).

Case 1 Memory-Bound 35M:21C					
bwaves (m) *1	bzip2 (c) *1	cactusADM (c) *1	calculix (c) *1	dealII (c) *1	gcc (m) *1
GemsFDTD (m) *2	gobmk (c) *4	gromacs (c) *1	h264ref (c) *1	hmmer (c) *3	lbm (m) *2
leslie3d (m) *2	libquantum (m) *6	mcf (m) *1	milc (m) *5	namd (c) *2	omnetpp (m) *5
povray (c) *3	sjeng (c) *1	soplex (m) *6	sphinx3 (m) *2	tonto (c) *2	xalancbmk (m) *2
Case 2 Half-Half 28M:28C					
astar (c) *1	bwaves (m) *1	bzip2 (c) *1	cactusADM (c) *1	calculix (c) *1	dealII (c) *1
gcc (m) *1	GemsFDTD (m)*1	gobmk (c) *4	gromacs (c) *1	h264ref (c) *1	hmmer (c) *4
lbm (m) *2	leslie3d (m) *1	libquantum (m) *5	mcf (m) *1	milc (m) *5	namd (c) *2
omnetpp (m) *4	povray (c) *4	sjeng (c) *1	soplex (m) *5	sphinx3 (m) *1	tonto (c) *4
wrf (c) *1	xalancbmk (m) *1	zeusmp (c) *1			
Case 3 Compute-Bound 21M:35C					
astar (c) *1	bwaves (m) *1	bzip2 (c) *1	cactusADM (c) *1	calculix (c) *2	dealII (c) *1
gcc (m) *1	GemsFDTD (m) *1	gobmk (c) *1	gromacs (c) *6	h264ref (c) *1	hmmer (c) *6
lbm (m) *1	leslie3d (m) *1	libquantum (m) *3	mcf (m) *1	milc (m) *3	namd (c) *2
omnetpp (m) *4	povray (c) *5	sjeng (c) *1	soplex (m) *3	sphinx3 (m) *1	tonto (c) *5
wrf (c) *1	xalancbmk (m) *1	zeusmp (c) *1			

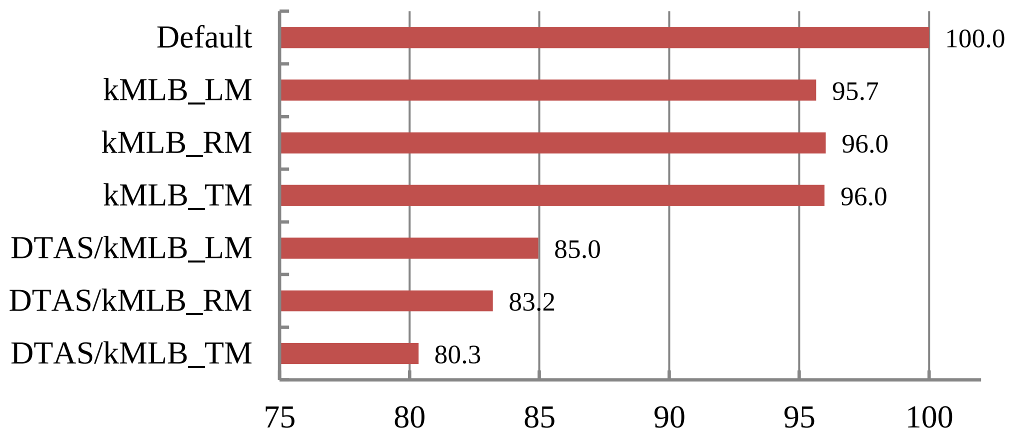
We measure the elapsed time for running benchmarks on seven Linux systems with different load balancing mechanisms. These systems are listed in Table 4. The default Linux kernel (i.e. Default) uses the unmodified Linux kernel with the default Linux kernel scheduling mechanism. The kMLB related systems use the modified Linux kernel with the kMLB mechanism and specific task selection policies. The DTAS/ kMLB related systems use the modified Linux kernel with the DTAS and the kMLB mechanisms.

Table 4. Various experimental systems

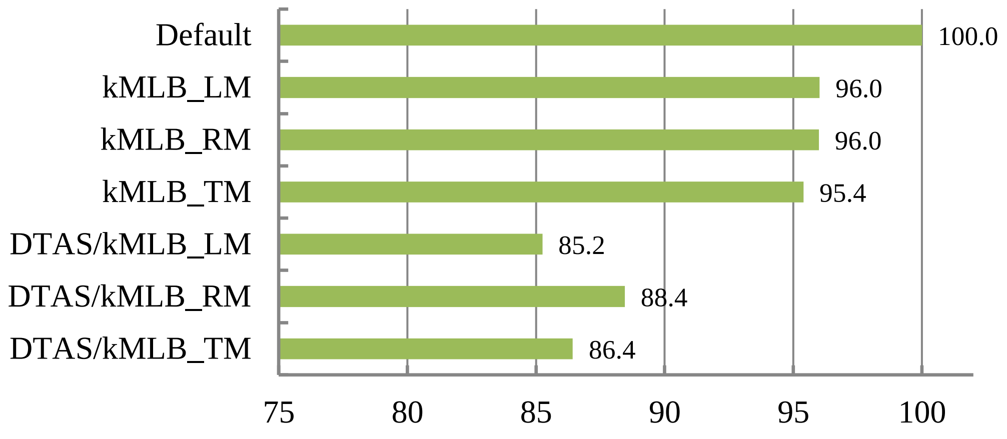
Notation	**Description**
Default	The default Linux kernel.
kMLB_TM	The modified Linux kernel with the kMLB mechanism and the Total Min policy.
kMLB_RM	The modified Linux kernel with the kMLB mechanism and the Remote Min policy.
kMLB_LM	The modified Linux kernel with the kMLB mechanism and the Local Max policy.
DTAS/kMLB_TM	The modified Linux kernel with the DTAS mechanism. The kMLB mechanism with the Total Min policy is incorporated.
DTAS/kMLB_RM	The modified Linux kernel with the DTAS mechanism. The kMLB mechanism with the Remote Min policy is incorporated.
DTAS/kMLB_LM	The modified Linux kernel with the DTAS mechanism. The kMLB mechanism with the Local Max policy is incorporated.

4.3 Experiment Results

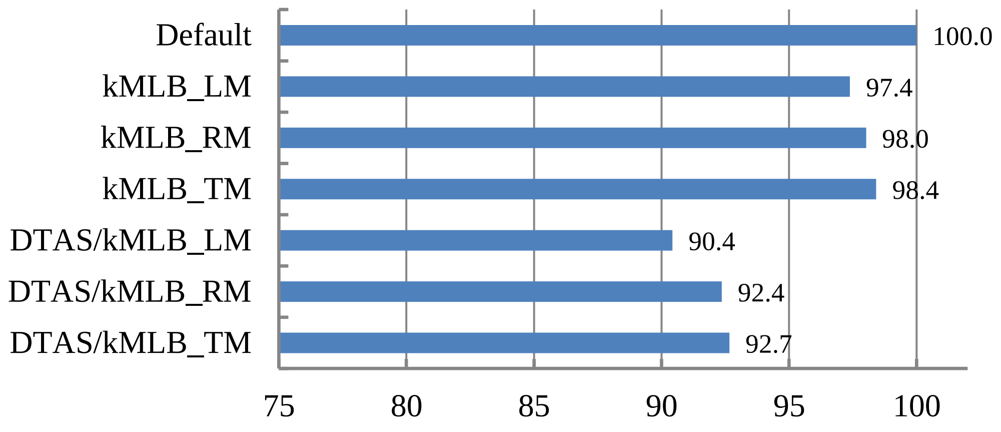
Figure 17 shows the experiment results. The elapsed time is the measured total execution time for running benchmarks. The x-axis shows the average elapsed time ratio of various experimental systems relative to the Default system that uses the default Linux kernel scheduling mechanism. If the ratio is larger than 100%, it means the performance of the specific experimental system is worse than that of the Default system.



(a) Case 1: memory-bound case (35M:21C).



(b) Case 2: half-half case (28M:28C).



(c) Case 3: compute-bound case (21M:35C).

Fig. 17. Performance compared to default Linux scheduling.

As shown in Figure 17, in all cases, the systems applying the kMLB mechanism have more prominent performance than that of the default Linux system due to the enhanced memory-aware task migration mechanism. This is because when a task is migrated across nodes, accessing memory on the previous node incurs remote memory access latency. With the kMLB mechanism, the scheduler is able to select suitable tasks for inter-node migration to balance CPUs' loading. kMLB is designed to select tasks that would incur less remote memory access for migration across nodes. For cases in which tasks request more memories, the systems with the kMLB mechanism obtain more benefit from our task selection policies. As a result, the kMLB systems perform the best in the memory-bound test case.

When the proposed kMLB mechanism is applied with the DTAS system, the system performance is further improved greatly. This is because the DTAS avoids resource contention while scheduling tasks on processor cores across nodes. Moreover, the DTAS mechanism automatically adjusts a processor's type according to the ratio of compute-bound tasks to memory-bound tasks currently running in the system. Therefore, among all of the experimental systems, the systems applying DTAS and kMLB mechanisms together have the most prominent performance.

Table 5. Performance improvement of kMLB over default Linux kernel.

Task selection policy	**Best case**	**Worst case**	**Average**
Case 1: Memory-bound case (35M:21C)			
kMLB (Total Min)	15.54 %	-6.79 %	4.02 %
kMLB (Remote Min)	8.76 %	-7.25 %	3.98 %
kMLB (Local Max)	15.64 %	-9.64 %	4.35 %
Case 2: Half-half case (28M:28C)			
kMLB (Total Min)	9.38 %	-4.48 %	4.61 %
kMLB (Remote Min)	10.46 %	-4.07 %	4.01 %
kMLB (Local Max)	12.88 %	-8.55 %	3.98 %
Case 3: Compute-bound case (21M:35C)			
kMLB (Total Min)	8.13 %	-2.64 %	1.58 %
kMLB (Remote Min)	11.44 %	-4.66 %	1.98 %
kMLB (Local Max)	13.80 %	-3.87 %	2.61 %

Tables 5 and 6 summarize the performance evaluation results of our kMLB mechanism. They show the elapsed time ratios of kMLB with different policies and DTAS/kMLB with different policies compared to those with the default Linux kernel scheduling. The "Best case" is the best elapsed time ratio and the "Worst case" is the worst elapsed time ratio when running a benchmark instance. The "Average" is the average elapsed time ratio of all benchmarks in a test case.

The performance improvement occurs because the kMLB mechanism considers tasks' memory usage on all nodes and selects suitable tasks for migration to balance CPUs' loading. It improves tasks' memory affinity by placing them on suitable nodes so as to reduce the need for tasks to access memories on other nodes. As a result, the systems with kMLB have more prominent performance in the memory-bound case than in the compute-bound case. Table 5 shows that when the kMLB mechanism is applied, performance improves by up to 15.64% in test case 1, 12.88% in test case 2, and 13.80% in test case 3. Among all policies, the Local Max policy has the most prominent improvement ratio.

Table 6. Performance improvement of DATS with kMLB over default Linux kernel.

Task selection policy	**Best case**	**Worst case**	**Average**
Case 1: Memory-bound case (35M:21C)			
DTAS/kMLB (Total Min)	46.64 %	-0.84 %	19.65 %
DTAS/kMLB (Remote Min)	49.92 %	-6.54 %	16.79 %
DTASkMLB (Local Max)	49.19 %	-17.27 %	15.04 %
Case 2: Half-half case (28M:28C)			
DTAS/kMLB (Total Min)	40.64 %	-8.08 %	13.57 %
DTAS/kMLB (Remote Min)	42.90 %	-15.91 %	11.55 %
DTAS/kMLB (Local Max)	40.60 %	-13.83 %	14.75 %
Case 3: Compute-bound case (21M:35C)			
DTAS/kMLB (Total Min)	19.21 %	-7.74 %	7.34 %
DTAS/kMLB (Remote Min)	16.72 %	-7.53 %	7.64 %
DTAS/kMLB (Local Max)	27.92 %	-7.93 %	9.57 %

Table 6 shows that when the systems are further applied with kMLB and DTAS together, the performance improves up to 49.92% in test case 1, 42.90% in test case 2, and 27.92% in test case 3. In these cases, most of the time the memory-bound tasks access memories on the local node and compete less for the shared resources. Some compute-bound tasks may wait longer for execution, because the amount of compute-bound tasks is more than the amount of compute-bound processors. The compute-bound tasks compete with each other for CPUs and slow down execution. Though the DTAS mechanism periodically adjusts processor type according to the ratio of compute-bound to memory-bound tasks, it may not rebalance the CPU type ratio in real time. On average, the improvement ratio of the Local Max policy is 9.57-15.04%, while the improvement ratio of the Remote Min policy and the Total Min policy are 7.64-16.79% and 7.34-19.65%, respectively.

According to the experiment results, our kMLB mechanism successfully improves system performance by decreasing tasks' need for remote memory access. The Linux kernel is modified to separately count the RSS value and keep track of each task's memory usage on each node. Despite this overhead, running tasks on the systems with kMLB mechanism is more efficient, regardless of whether the DTAS mechanism is used. Among all cases, the systems with DTAS and kMLB mechanisms have the best performance, since these mechanisms reduce resource contention and remote memory access at the same time.

5 CONCLUSION AND FUTURE WORKS

Both inter-node memory access and resource contention decrease system performance on NUMA systems. Moreover, the kernel scheduler's migrating of tasks across nodes for load balancing further increases remote memory access, since tasks' allocated memories spread on multiple nodes. Remote memory access also increases contention for inter-node interconnect links. Though the Linux kernel has implemented automatic page migration [12-14] which migrates the memory page of a process to the node it is currently running on, this migration is driven by page fault handling when the page is actually accessed. This increases the additional overhead for memory access due to the fault handling and page migration. Furthermore, page migration is not beneficial for occasionally accessed pages.

Because the cost of supporting inter-node page migration is high, we propose the kernel-level Memory-aware Load Balancing mechanism (kMLB) to enhance the Linux kernel scheduler's load balancing mechanism. This allows us to select suitable tasks for migration across nodes. The aim of the proposed memory-aware task selection policies is mainly to reduce memory migration frequency and remote memory access. In the implementation, the Linux kernel is modified to keep track of the memory usage of each task on each node. The kernel's load balancing procedure is modified to incorporate the proposed task selection policies. These policies use the memory usage information to choose the most suitable task for inter-node migration.

We have implemented the proposed mechanism and policies in the Linux kernel. The experiment results show that our kMLB mechanism improves the performance of kernel scheduling by reducing remote memory access and resource contention. We also found more improvement in performance when applying the Local Max memory policy, compared to other policies for avoiding tasks' remote memory access.

ACKNOWLEDGMENTS

This research was supported in part by grant MOST 104-2221-E-260-001- and 105-2221-E-260-015- from the Ministry of Science and Technology, Taiwan, Republic of China.

REFERENCE

[1] Symmetric Multiprocessing, https://en.wikipedia.org/wiki/Symmetric_multiprocessing, accessed on Jan. 17, 2017.

[2] Christoph Lameter, "An Overview of Non-Uniform Memory Access," Communications of the ACM, Vol. 56, Issue 9, pp. 59-65, September 2013.

[3] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A Case for NUMA-aware Contention Management on Multicore Systems," In Proceedings of the USENIX Annual Technical Conference, Berkeley, CA, USA, 2011.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing Shared Resource Contention in Multicore Processors via Scheduling," In Proceedings of the 15th Architectural Support for Programming Languages and Operating Systems, pp. 129-142, 2010.

[5] A. Merkel and F. Bellosa, "Memory-aware Scheduling for Energy Efficiency on Multicore Processors," In Proceedings of the Workshop on Power Aware Computing and Systems, pp. 123-130, San Diego, CA, USA, Dec. 2008.

[6] A. Merkel, J. Stoess, and F. Bellosa, "Resource-conscious Scheduling for Energy Efficiency on Multicore Processors," In Proceedings of the 5th European

Conference on Computing Systems, pp. 153-166, Paris, France, Apr. 2010.

[7] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto, "Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors," ACM Computing Surveys, Vol. 45, No. 1, pp. 1-28, Nov. 2012.

[8] M. L. Chiang, C. J. Yang, and S. W. Tu, "Kernel Mechanisms with Dynamic Task-Aware Scheduling to Reduce Resource Contention in NUMA Multicore Systems," Journal of Systems and Software, Vol. 121, pp. 72-87, Nov. 2016.

[9] Uniform Memory Access, https://en.wikipedia.org/wiki/Uniform_memory_access, accessed on Jan. 17, 2017.

[10] C. Terboven, D. an Mey, D. Schmidl, H. Jin, and T. Reichstein, "Data and Thread Affinity in OpenMP Programs," In Proceedings of the 2008 workshop on Memory access on future processors, pp. 377-384, 2008.

[11] B. Goglin and N. Furmento, "Enabling High-Performance Memory Migration for Multithreaded Applications on Linux," In Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, May 23-29, 2009.

[12] V. K. Mishra and D. A. Mehta, "Performance Enhancement of NUMA Multiprocessor Systems with On-Demand Memory Migration," In Proceedings of the 2013 IEEE 3rd International Advance Computing Conference, pp. 40-43, Feb. 22-23, 2013.

[13] Linux NUMA evolution, https://www.dcl.hpi.uni-potsdam.de/teaching/numasem/slides/NUMASem_Linux_NUMA_evolution.pdf, accessed on Jan. 17, 2017.

[14] Automatic NUMA Balancing, http://events.linuxfoundation.org/sites/events/files/slides/summit2014_riel_chegu_w_0340_automatic_numa_balancing_0.pdf, accessed on Jan. 17, 2017.

[15] Automatic Page Migration for Linux, ftp://mirror.internode.on.net/pub/linux.conf.au/2007/video/thursday/197.pdf, accessed on Jan. 17, 2017.

[16] Completely Fair Scheduler, https://en.wikipedia.org/wiki/Completely_Fair_Scheduler, accessed on Jan. 17, 2017.

[17] Scheduling domains, http://lwn.net/Articles/80911/, accessed on Jan. 17, 2017.

[18] Cpuset Management Utility, https://rt.wiki.kernel.org/index.php/Cpuset_Management_Utility/tutorial, accessed on Jan. 17, 2017.

[19] Procfs, https://en.wikipedia.org/wiki/Procfs, accessed on Jan. 17, 2017.

[20] Hardware Performance Counter,

http://en.wikipedia.org/wiki/Hardware_performance_counter, accessed on Jan. 17, 2017.

[21] R. Knauerhase, P. Brett, B. Hohlt, T. Li, and S. Hahn, “Using OS Observations to Improve Performance in Multicore Systems,” IEEE Micro, vol. 28, pp. 54-66, 2008.

[22] M. Dashti et al., “Traffic Management: a Holistic Approach to Memory Placement on NUMA Systems,” In Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 381-394, Mar. 16-20, 2013, Houston, Texas, USA.

[23] S. Srikanthan, S. Dwarkadas, and K. Shen, “Data Sharing or Resource Contention: Toward Performance Transparency on Multicore Systems,” In Proceedings of the 2015 USENIX Annual Technical Conference, July 8-10, 2015, Santa Clara, CA, USA.

[24] B. Lepers, V. Qu´ema, and A. Fedorova, “Thread and Memory Placement on NUMA Systems: Asymmetry Matters,” In Proceedings of the 2015 USENIX Annual Technical Conference, July 8-10, 2015, Santa Clara, CA, USA.

[25] Resident Set Size, https://en.wikipedia.org/wiki/Resident_set_size, accessed on Jan. 17, 2017.

[26] ASUS TS500-E6 Server, http://www.asus.com/Commercial_Servers_Workstations/TS500E6PS4/, accessed on Jan. 17, 2017.

[27] Z8NA-D6 motherboard, http://dlcdnet.asus.com/pub/ASUS/server/TS500-E6_PS4/Manual/T4656_TS500-E6_PS4.pdf, accessed on Jan. 17, 2017.

[28] Intel® Hyper-Threading Technology, https://en.wikipedia.org/wiki/Hyper-threading, accessed on Jan. 17, 2017.

[29] SPEC CPU2006, https://www.spec.org/cpu2006/, accessed on Jan. 17, 2017.

***Title page with author details**

Kernel-level Memory-Aware Load Balancing Mechanism on NUMA Multicore Systems

Mei-Ling Chiang[1] Shu-Wei Tu[2]

Department of Information Management,
National Chi Nan University,
Nantou, Puli, Taiwan, R.O.C.
Email: {joanna[1], s101213531[2]}@ncnu.edu.tw