applied sciences

Article

Thread-Aware Mechanism to Enhance Inter-Node Load Balancing for Multithreaded Applications on NUMA Systems

Mei-Ling Chiang * and Wei-Lun Su

Department of Information Management, National Chi Nan University, Puli 54516, Taiwan; s94213034@gmail.com
* Correspondence: joanna@mail.ncnu.edu.tw

Abstract: NUMA multi-core systems divide system resources into several nodes. When an imbalance in the load between cores occurs, the kernel scheduler's load balancing mechanism then migrates threads between cores or across NUMA nodes. Remote memory access is required for a thread to access memory on the previous node, which degrades performance. Threads to be migrated must be selected effectively and efficiently since the related operations run in the critical path of the kernel scheduler. This study focuses on improving inter-node load balancing for multithreaded applications. We propose a thread-aware selection policy that considers the distribution of threads on nodes for each thread group while migrating one thread for inter-node load balancing. The thread is selected for which its thread group has the least exclusive thread distribution, and thread members are distributed more evenly on nodes. This has less influence on data mapping and thread mapping for the thread group. We further devise several enhancements to eliminate superfluous evaluations for multithreaded processes, so the selection procedure is more efficient. The experimental results for the commonly used PARSEC 3.0 benchmark suite show that the modified Linux kernel with the proposed selection policy increases performance by 10.7% compared with the unmodified Linux kernel.

Keywords: NUMA; Linux kernel; multithreaded; load balancing; remote memory access

check for updates

Citation: Chiang, M.-L.; Su, W.-L. Thread-Aware Mechanism to Enhance Inter-Node Load Balancing for Multithreaded Applications on NUMA Systems. *Appl. Sci.* **2021**, *11*, 6486. https://doi.org/10.3390/app11146486

Academic Editor: Juan-Carlos Cano

Received: 8 June 2021
Accepted: 12 July 2021
Published: 14 July 2021

1. Introduction

Multi-core systems allow parallel computing and have a higher throughput. To effectively utilize the performance of multi-cores, applications are coded as multithreaded. In Linux, the kernel maintains one runqueue for each core. When a process or thread is ready to run, it is put into the runqueue and waits to be run on the corresponding core. The Linux kernel [1] maintains a data structure, struct task_struct, which records attributes and runtime information for each schedulable entity. Each schedulable entity in the Linux kernel is called a task. When several tasks with different run times are run simultaneously, the load between cores can be imbalanced, so performance is decreased. The kernel scheduler's load balancing mechanism then migrates tasks from the overloaded core's runqueue to the runqueue of a core that is not so heavily loaded.

Non-Uniform Memory Access (NUMA) [2] systems divide system resources such as processors, caches, and RAM into several nodes. It takes longer for one core to access the memory on different NUMA nodes than on the local node. This costly memory access is called remote memory access. For a task running on a NUMA system, the memory pages allocated to it may be scattered on different nodes. When a task accesses memory pages on nodes other than those on which it runs, remote memory access occurs. For Linux-based NUMA systems, the load balancing mechanism can migrate tasks to another node, so costly remote memory access is necessary after the migration. The benefit of load balancing is reduced by the need for remote memory access after task migration. A prior study [3] showed that reducing remote memory access is critical in designing and implementing an operating system on NUMA systems.

In order to maintain a balanced load and reduce remote memory access, the kernel-level Memory-aware Load Balancing (kMLB) [4] mechanism was proposed to allow better inter-node load balancing for the Linux-based NUMA systems. The unmodified Linux kernel migrates the first movable task that can be run on the target core. The task may involve more remote memory access when it is migrated between nodes. The kMLB mechanism modifies the Linux kernel to track each task's number of memory pages on each node. This memory usage information is then used to select the task that is most suited to inter-node migration. Several task selection policies, such as Most Benefit (MB) [4] and Best Cost-Effectiveness [4], have been proposed and used different metrics. The selected tasks require less remote memory access after migration, and system performance successfully improves. Differently, Chen et al. [5] proposed a machine learning-based resource-aware load balancer in the Linux kernel to make migration decisions. The extra runtime overhead deducts the performance gain because scheduling operation is in the critical path of kernel operations.

A multithreaded process can create threads as needed during its execution. In the Linux kernel, these threads form one thread group and share memory space. On NUMA systems, threads of one thread group can be scheduled by the kernel to run on different nodes to balance the load, so the same memory pages can be accessed by threads that run on different nodes. Accessing one memory page involves local access for some threads and remote access for other threads. When a thread is migrated across nodes, remote memory access and cache misses increase as well, so it is difficult to determine the cost of memory access.

This study first analyzes multithreaded applications and their memory access in Linux and then proposes a new task selection policy, which is named Exclusivity (Excl) for multithreaded applications. This policy determines whether a task is suitable for inter-node migration using the exclusivity of the thread distribution on NUMA nodes in its thread group. The task for which the thread group is least exclusive is selected, which has a lesser effect on data mapping and thread mapping for its thread group.

Although selecting a suitable task for inter-node migration can reduce remote memory access, the selection procedure must evaluate all tasks in the runqueue, which involves a processing overhead. Since the selection is in the critical path for the kernel scheduler, the cost of more operations outweighs any benefit. Therefore, we further improve the procedure for selecting tasks by using thread group information to eliminate superfluous evaluations. Only a subset of movable tasks in the runqueue is evaluated, so the selection procedure is more efficient.

The contribution of this study is as follows. First, multithreaded applications and their memory access in Linux are analyzed. The analysis indicates that the existing memory-aware MB policy is still effective for multithreaded applications. However, it requires the kMLB mechanism to track per-task memory usage on per NUMA node. Thus, its implementation needs to modify many kernel operations and data structures that are affected. Second, the thread-aware Exclusivity policy is proposed, which is a relatively lightweight task selection policy since it does not need the kMLB mechanism. Instead, it uses the exclusivity of the thread distribution on nodes in the thread group to determine the target thread for inter-node migration. Third, several methods to enhance selecting tasks for inter-node migration for multithreaded applications are proposed.

Finally, the proposed Exclusivity policy and enhancement methods are practically implemented in the Linux kernel. Extensive experiments using the PARSEC 3.0 [6] benchmark suite run on the modified Linux kernel with various task selection policies. Compared with the unmodified Linux kernel, the results show that when the task selection procedure is enhanced, the Most Benefit Plus (MB$^+$) policy, which requires the kMLB mechanism, increases performance by 11.1%. The proposed Exclusivity policy increases performance by 10.7%. This policy is competitive and does not require the kMLB mechanism. Besides, it is more easily adapted to a newer Linux kernel.

The remainder of this paper is organized as follows. Section 2 introduces the technological background and related work. Section 3 presents the improvements for inter-node task migration for multithreaded applications and the new task selection policy. Section 4 details the experimental results, and Section 5 concludes.

2. Technological Background and Related Work

Though multi-core systems have high throughput, the performance increase depends on the placement of tasks and their data on nodes during runtime. Task placement affects contention between resources for the cache and cores, and data placement affects the memory access cost for a task. To utilize system resources more efficiently and increase performance, many studies [7–15] design specific placements of tasks and data to decrease resource contention, balance the loads in the cores, and reduce access to remote memory. The solutions focus on improving the performance of Symmetric Multi-processing (SMP) and Non-Uniform Memory Access (NUMA) [2] systems.

However, the rises of multithreaded applications cause finding specific task placement or data placement to increase performance more complicated. Since threads in one multithreaded application commonly share memory address space, and if they run on cores that do not share or share fewer cache resources, tasks are placed less efficiently because of more cache misses. In terms of data placement, there are different memory access latencies for cores access memory on different nodes. It is more challenging to determine where to allocate the required memory for the requesting task on NUMA systems.

This section first describes two types of multi-core systems and then reviews related work in Section 2.2. Section 2.3 introduces the kernel-based Memory-aware Load Balancing (kMLB) [4] mechanism and task selection policies proposed for improving inter-node migration.

2.1. Multi-Core Systems

A Symmetric Multi-processing (SMP) system is a multi-core system in which cores can share different levels of cache and the cost to access any location in the main memory is the same for all cores, as shown in Figure 1a. For a Non-Uniform Memory Access (NUMA) [1] system, as shown in Figure 1b, system resources such as cores, RAM, and memory controllers are divided into several nodes, and interconnection links connect these nodes. Cores can access the memory located on the same node with it, which is called local memory access. Accessing memory on other nodes via interconnection links is called remote memory access. This requires a longer time. The NUMA factor is the ratio between the remote memory access latency and the local memory access latency.

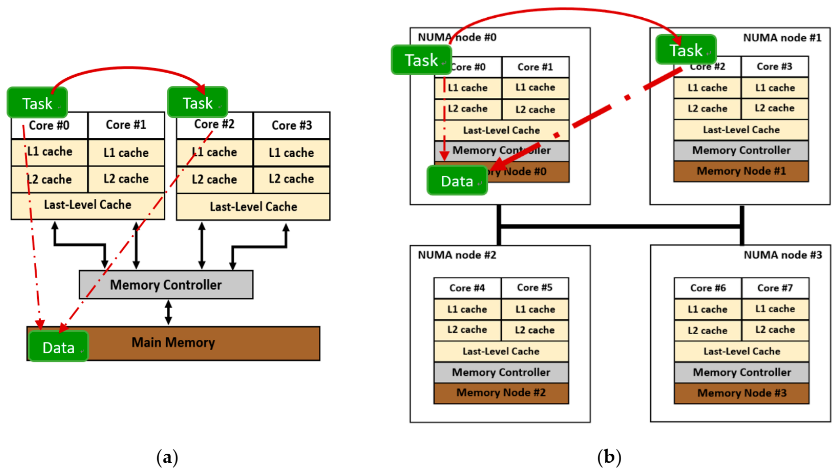


Figure 1. The load balancing mechanism affects SMP and NUMA systems differently. (**a**) SMP system; (**b**) NUMA system.

Though a NUMA system is more scalable than an SMP system and more memory accesses can occur simultaneously, operating system design must reduce costly remote memory access. Nevertheless, for load balancing, modern operating systems migrate tasks from an overloaded core's runqueue to the runqueue of a core with a lesser load. The load-balancing mechanism affects SMP and NUMA systems differently, as shown in Figure 1. Unlike an SMP system that features a uniform cost to access memory, migrating a task across nodes can involve costly remote memory access after migration. The benefit gained from load balancing thus is reduced by the need for remote memory access.

2.2. Related Work

In the Linux kernel, a memory page is allocated to a task when it first accesses the page. The page is allocated on the node where the requesting task is running. This is called a first-touch strategy [16]. Suppose a task is scheduled to run on another NUMA node during context switches or is migrated to another node for load balancing. In that case, the task requires remote memory access to access the page on the original node. Linux [17] offers NUMA-related system calls for NUMA-aware programs and provides commands and tools that constrain tasks to run on specific nodes. Several studies [11–14] proposed methods that use NUMA-related system calls to bind one task on specific nodes to decrease remote memory access. However, these performance improvements are reduced because a multithreaded application can create threads as needed during runtime, and the thread load is not deterministic. Binding tasks on specific nodes results in a load imbalance between nodes, and CPU utilization decreases accordingly.

Chen et al. [5] implemented a machine learning (ML)-based resource-aware load balancer in the kernel to make migration decisions. An ML model is implemented inside the kernel to monitor real-time resource usage in the system. This identifies potential hardware performance bottlenecks and then makes load balancing decisions. This ML model is trained offline in the user space and is used for online inference in the kernel to generate migration decisions. The results show no significant difference in the performance of the original kernel and the modified kernel when running benchmarks. Performance gains are negated largely because of the extra runtime overhead and the fact that scheduling operations are in the critical path of kernel operations.

Migrating memory pages to the NUMA node on which their requesting task is currently running reduces remote memory access. Mishra and Mehta [15] proposed an on-demand memory migration policy that migrates only the referenced pages to the current node where the requesting task is running. Terboven et al. [11] proposed a user-level implementation of a Next-touch approach in Linux. The *mprotect()* system call is used to change protection on a memory region, so the successive reads and writes incur segmentation faults. A signal handler that handles segmentation fault is implemented and invokes the *move_pages()* system call to migrate the accessed page to the node on which the task is currently running. Goglin and Furmento [13,14] presented two different implementations of a Next-touch approach in Linux. The user-space implementation also uses the *mprotect()* system call and a segmentation fault signal handler to migrate the accessed page. The kernel-level implementation uses the *madvise()* system call and modifies the kernel page fault handler to migrate the accessed page. The results show that the kernel-based implementation is more efficient than the user-space implementation. However, if the memory page is not accessed again after it is migrated, the cost of accessing the remote memory page may be less than the cost of migrating it.

In the Linux kernel, all threads of one multithreaded process share memory address space and use the same page table. If these threads run on different nodes, it is hard to determine whether pages should be migrated to the node where the requesting thread runs and to track the memory access pattern for an individual thread. Therefore, it is more challenging to perform thread mappings or data mappings to reduce remote memory access. To overcome these difficulties, Diener et al. [7] modified the kernel page fault handling routines to track the memory access patterns for any threads. The present flag of the page table entry is cleared so that whenever one memory page is accessed, a page fault occurs, even though the faulted memory page has already been in the memory. This identifies which thread on which node accesses this memory page and its access pattern. This mechanism is named kMAF [7] and uses the memory access patterns to determine which threads are more relevant and migrates them to the same node to allow better thread mappings. For data mappings, kMAF migrates one memory page to the node where the frequency of faults for this page is exclusive, so the page is mostly accessed from that node. This reduces remote memory access.

For methods using page fault to trigger migrating the faulted page to the same node as the faulting thread, the induced faults reduce performance. Existing studies also use page faults on the same page table for all threads for one multithreaded process to determine the memory access pattern for data mappings or thread mappings. Since several threads of a multithreaded process may fault one page, Gennaro et al. [8] indicated that this might result in an inaccurate estimation of the working-set of individual threads for one multithreaded process performance is decreased in terms of thread mappings. They then proposed a solution that uses the multi-view address space (MVAS). If MVAS is switched on while one multithreaded process runs, one individual page table is created for each of its threads. The memory access pattern for different threads can be separately tracked in different page tables until MVAS is switched off for the multithreaded process. MVAS does not incur extra page faults, so it can support those studies [11,13,14] that use page faults to perform page migrations to reduce remote memory access.

Lepers et al. [9] studied the placement of threads and data on NUMA nodes and the asymmetry of interconnecting links for nodes connected by links of different bandwidths. A dynamic thread and memory placement algorithm was developed in Linux to minimize contention for asymmetric interconnect links and maximize bandwidth between communicating threads. Li et al. [10] also studied the effect of hardware asymmetry. The AMPS scheduler is implemented in the Linux kernel to support asymmetric multicore architectures for which cores in the same processor have different performances. The Linux kernel is modified to track the memory usage for each thread on each node and predicts the migration overhead for a thread. Threads are migrated to faster cores when they are under-utilized. However, if the predicted migration overhead is too high or the thread is in the memory allocation phase, this thread cannot be migrated across nodes.

2.3. The Kernel-Based Memory-Aware Load Balancing (kMLB) Mechanism

Inter-node task migration is necessary for an operating system to balance the load between NUMA nodes, and migrating different tasks for inter-node load balancing incurs a different amount of remote memory access. As shown in Figure 2, migrating the first task in the source runqueue, i.e., Task#3, incurs 3-page remote memory access but migrating Task#10 incurs 5-page remote memory access. However, the unmodified Linux kernel always selects the first task that can be migrated to run on the destination core. It allows rapid selection, but the first task may not incur the least remote memory access after the inter-node migration.

A previous study [4] shows that it is better to select the task that involves less remote memory access after migration. The kernel-based Memory-aware Load Balancing (kMLB) mechanism was proposed to select suitable tasks to migrate between nodes to allow better load balancing in the Linux kernel. The memory usage for each task on each node is tracked. Depending on a task's current running node, the physical pages that it occupies on each node are identified as local or remote. The load balancer then uses this information to determine the most suitable task for inter-node migration.

In the Linux kernel, the Resident Set Size (RSS) [1] for one process is the number of page frames occupied by this process. The original RSS only tracks the total number of page frames that are occupied by each process. The kMLB mechanism modifies the kernel operations to track the RSS counters on each node for each process, including dynamic memory allocation and releases, demand paging, swapping, system calls, and inter-node page migration.

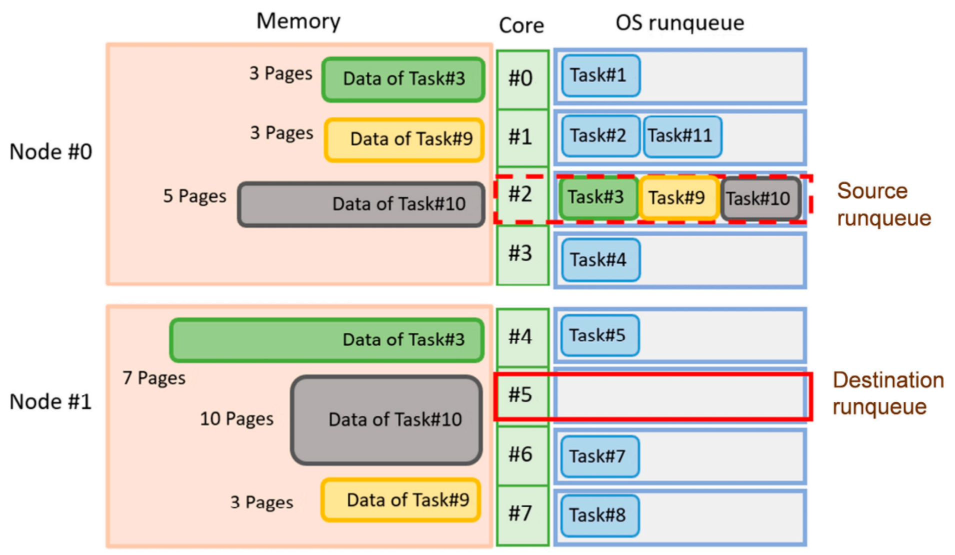


Figure 2. Migrating different tasks incurs a different degree of remote memory access after migration.

The following details task selection policies that are used with the kMLB mechanism. For the example in Figure 2, Table 1 shows the selected task for each policy according to the specific metric used.

Table 1. Selecting a task using different selection policies.

Policy	Tasks Waiting in the Source Core's Runqueue			Selected Task
	Task#3	Task#9	Task#10	
First-Fit	N/A	N/A	N/A	Task#3
TM	3 + 7 = 10	3 + 3 = 6	5 + 10 = 15	Task#9
MB	7 − 3 = 4	3 − 3 = 0	10 − 5 = 5	Task#10
BCE	(7 − 3)/3 = 1.333	(3 − 3)/3 = 0	(10 − 5)/5 = 1	Task#3

- Total Min (TM) Policy [3] selects the task in the source core's runqueue with the minimal total memory size. It is possible to have the least influence caused by task migration since the selected one occupies the least amount of memory for access after migration.
- Most Benefit (MB) Policy selects the task in the source core's runqueue that can reduce the maximum amount of remote memory access when migrated. This policy considers the memory that is occupied by each task on the source and destination nodes. The task with the maximum difference is selected. The following metric is used to select the target task with the maximum value, in which $\mathrm{RSS}_p(i)$ is the RSS value for task p on NUMA node i and *dest_node* and *src_node* are the IDs for the destination and source NUMA nodes, respectively:

$$\mathrm{MB}_p = \mathrm{RSS}_p(dest\_node) - \mathrm{RSS}_p(src\_node)$$

- Best Cost-Effectiveness (BCE) Policy selects the task in the source core's runqueue for which inter-node migration is the most cost-effective. Based on the MB policy, the BCE policy also considers the maximum cost of page migration, which is the amount of memory occupied by one task on the source node. The selected task is the one that can reduce the maximum remote memory access relative to the maximum migration cost when it is migrated. The following metric is used to select the target task with the maximum value:

$$\mathrm{BCE}_p = (\mathrm{RSS}_p(dest\_node) - \mathrm{RSS}_p(src\_node))/\mathrm{RSS}_p(src\_node)$$

3. Improved Inter-Node Load Balancing for Multithreaded Applications

The kMLB mechanism [4] tracks the number of physical pages per node occupied by each task. For each movable task in the overloaded core's runqueue, the modified OS scheduler uses this information to calculate the metric to determine the most suitable task for inter-node migration. However, the kernel scheduler must evaluate each task in the runqueue to identify a target task to be migrated. The source (i.e., the busiest) and the destination (i.e., the idlest) runqueues must be locked, so the target task must be identified efficiently because it runs in the critical path of the kernel scheduler. Besides, the threads of one multithreaded application share memory pages and may be distributed on different nodes. When threads running on different nodes access their shared memory pages on the remote node, cache misses and remote memory access slow access.

This study improves inter-node task migration for multithreaded applications in two respects. This section first introduces multithreaded applications and their memory access in Linux. Section 3.2 describes the improvements in selecting tasks for migration between nodes for multithreaded applications. Section 3.3 presents the proposed thread-aware task selection policy for inter-node migration for multithreaded applications.

3.1. Multithreaded Applications and Their Memory Access in Linux

During the execution of a multithreaded application, threads are created as needed. In Linux, these threads form one thread group. The first thread in a multithreaded process is the thread group leader, and other threads are the members of this thread group. Each thread is regarded as one schedulable entity in the Linux kernel, so it is one task. Threads of the same thread group share the same memory address space and page table.

When a task is created, the OS scheduler dispatches it to the core with the least load to maintain load balance within multi-core systems. On NUMA systems, threads of the same thread group may be distributed on different nodes, and their memory pages can also be allocated on several nodes, as shown in Figure 3. Therefore, memory pages are local memory pages for some threads and remote memory pages for others. Access to Data0 is local access for threads T0, T1, and T7 and remote access for threads T2, T3, T4, T5, and T6. The difference in the total memory access cost for a thread group when one of its threads is migrated across nodes must be determined.

As depicted in Figure 4, threads within one thread group can be scheduled to run on different nodes, and their memory spaces can be on different nodes. A thread group's total memory access cost sums up the local memory access costs and the remote memory access costs for threads in this thread group. For a NUMA system with n nodes for which the memory access latency from a remote node to the local node is constant, the number of threads within a specific thread group on the NUMA node i is denoted as N_i. The number of memory pages allocated to this thread group on the NUMA node i is denoted as R_i.

Regardless of the locality of memory references, the latency for all threads in the thread group to access the entire memory allocated to the thread group, which is the estimated total memory access cost, is shown in Equation (1). The local memory access cost is shown in Equation (2). Equation (3) shows the remote memory access cost, in which f is the NUMA factor that represents the ratio between the remote memory access latency and the local memory access latency.

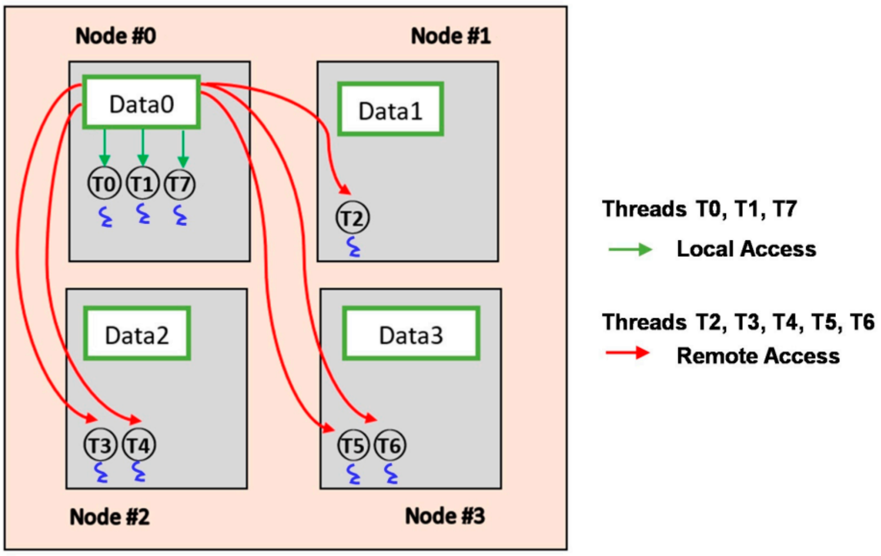


Figure 3. The memory pages and threads for a multithreaded process can be scattered on different nodes.

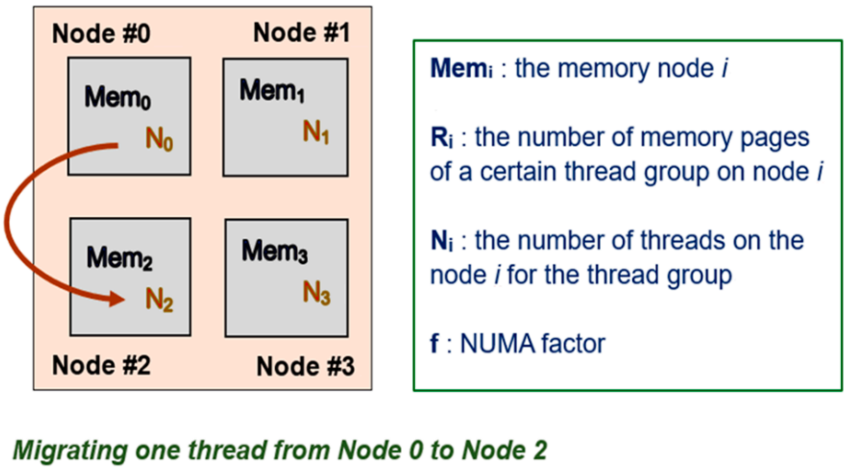

Node #0
Node #1
Mem₀
N₀
Mem₁
N₁
Mem₂
N₂
Mem₃
N₃
Node #2
Node #3
Memᵢ : the memory node i
Rᵢ : the number of memory pages of a certain thread group on node i
Nᵢ : the number of threads on the node i for the thread group
f : NUMA factor
Migrating one thread from Node 0 to Node 2

Figure 4. Migrating one thread of a specific thread group across nodes.

$$\text{Total memory access cost (TMA) = Local memory access cost (LMA) + Cache miss cost + Remote memory access cost (RMA)} \tag{1}$$

$$\text{Local memory access cost (LMA)} = \sum_{i=0}^{n-1} N_i * R_i \tag{2}$$

$$\text{Remote memory access cost (RMA)} = \left(\sum_{i=0}^{n-1} \sum_{\forall\, j \neq i} N_i * R_j\right) * f \tag{3}$$

Therefore, for the case in Figure 4, the estimated total memory access cost for this thread group is the value that is shown in Equation (4):

$$\begin{aligned} \text{TMA} &= \textstyle\sum_{i=0}^{3} N_i * R_i + \text{Cache miss cost} + \left(\sum_{i=0}^{3} \sum_{\forall\, j \neq i} N_i * R_j\right) * f \\ &= N_0R_0 + N_1R_1 + N_2R_2 + N_3R_3 + \text{Cache miss cost} + \\ &(N_0R_1 + N_0R_2 + N_0R_3 + N_1R_0 + N_1R_2 + N_1R_3 + N_2R_0 + N_2R_1 + N_2R_3 + N_3R_0 + N_3R_1 + N_3R_2) * f \end{aligned} \tag{4}$$

Because only the values of N_0 and N_2 change, and the others remain the same, the difference between the total memory access cost after a thread is migrated from node 0 to node 2 is simplified and calculated using Equation (5):

$$\begin{aligned}\text{Difference} &= \text{TMA (after the migration of one thread from node 0 to node 2)} - \text{TMA (before the migration)} \\ &= (R_2 - R_0) * (1 - f)\end{aligned} \tag{5}$$

That is, if one thread is migrated, the difference between the total memory access cost after the migration is calculated using Equation (6), where R_D is the RSS value in the destination node, R_S is the RSS value in the source node, and f is the NUMA factor:

$$\text{Difference} = (R_D - R_S) * (1 - f) \tag{6}$$

Regarding inter-node task migration, the RSS values for a thread group on the source and the destination nodes have the most significant effect on the total memory access cost. Therefore, Most Benefit (MB) [4], which uses the same metric to select the most beneficial task is also appropriate for multithreaded applications.

3.2. Enhancements for Selecting Tasks for Inter-Node Migration for Multithreaded Applications

Selecting a suitable task for inter-node migration requires additional overhead because the selection procedure must evaluate all tasks in the runqueue in order. The evaluation cost increases as the number of tasks in the runqueue increases. For multithreaded applications, some evaluations are superfluous and can be eliminated because some tasks are less suitable than the candidate task. In this study, the thread group leaders are not migrated. The thread group leader's current node is determined when its thread group members are evaluated, and only one thread member per thread group is evaluated. Some tasks for the evaluation are eliminated if they match one of these aspects. Therefore, only the subset of movable tasks in the runqueue is evaluated, and the procedure for selecting tasks is more efficient. Figure 5 shows the flow for the improvements, and these methods are explained in the following subsections.

3.2.1. Eliminating the Thread Group Leader for Migration

The thread group leader is not selected because the Linux kernel uses the first-touch method [16] for physical memory allocation. A physical memory page is allocated the first time a thread accesses it and on the same node as the requesting thread. We observe the memory consumption for multithreaded applications, the first thread that touches the memory page is usually the thread group leader. The PARSEC 3.0 [6] benchmark suite contains 30 multithreaded applications, including 13 programs from PARSEC 2.0, 14 programs from the SPLASH benchmark suite, and three network programs. Table 2 shows these benchmark programs [4,6].

Each benchmark program's memory consumption was measured during its execution on the AMD server [18]. In Linux, the *free* command is used to obtain the current status for memory usage. A script that executes the free command every 0.1 s is used to record the memory usage footprint for the entire system. The increased memory usage during the benchmark program's execution is then attributed to the benchmark program. Each benchmark program is run with different threads, ranging from 1, 2, 4, 8, 16, and 32.

The results show that memory consumption is independent of the number of created threads for some benchmark programs. Other benchmark programs consume more memory as the number of created threads increases. Figure 6a shows the former situation for the benchmark program parsec.canneal. For this type of benchmark program, the memory pages of a multithreaded process are first touched by the first thread. For the benchmark program splash2x.fmm in Figure 6b, the individual threads first touch the memory pages of a multithreaded process. The results in Table 2 show that 22 of 30 applications allocate and initialize the allocated memory pages in the initializing thread (i.e., the thread group leader). The thread group leader for each multithreaded process is not migrated to reduce the scattering of memory pages on different nodes.

Table 2. The characteristics of benchmark programs in PARSEC 3.0.

Benchmark Suite	Benchmarks	Memory Allocated & Initialized
PARSEC 2.0	blackscholes, bodytrack, canneal, dedup, raytrace, streamcluster, swaptions	by thread group leader
	facesim, ferret, fluidanimate, freqmine, vips, x264	by individual threads
SPLASH-2x	barnes, cholesky, fft, lu_cb, lu_ncb, ocean_cp, ocean_ncp, radiosity, radix, raytrace, volrend, water_spatial	by thread group leader
	fmm, water_nsquared	by individual threads
Network	netdedup, netferret, netstreamcluster	by thread group leader

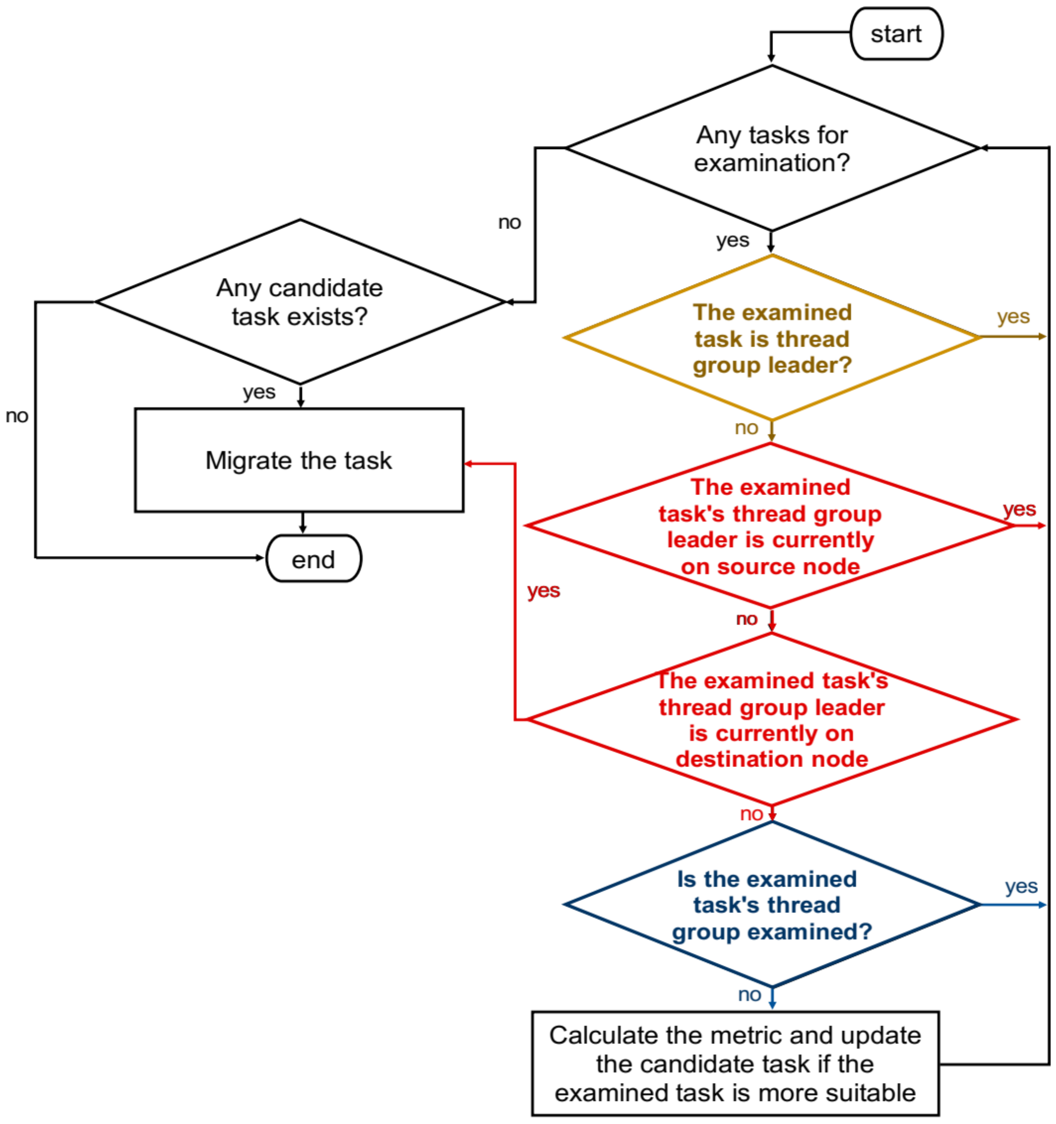

start
Any tasks for examination?
no
yes
Any candidate task exists?
yes
no
Migrate the task
end
The examined task is thread group leader?
yes
no
The examined task's thread group leader is currently on source node
yes
no
yes
The examined task's thread group leader is currently on destination node
no
Is the examined task's thread group examined?
yes
no
Calculate the metric and update the candidate task if the examined task is more suitable

Figure 5. The modified flow for the efficient selection of tasks for inter-node migration.

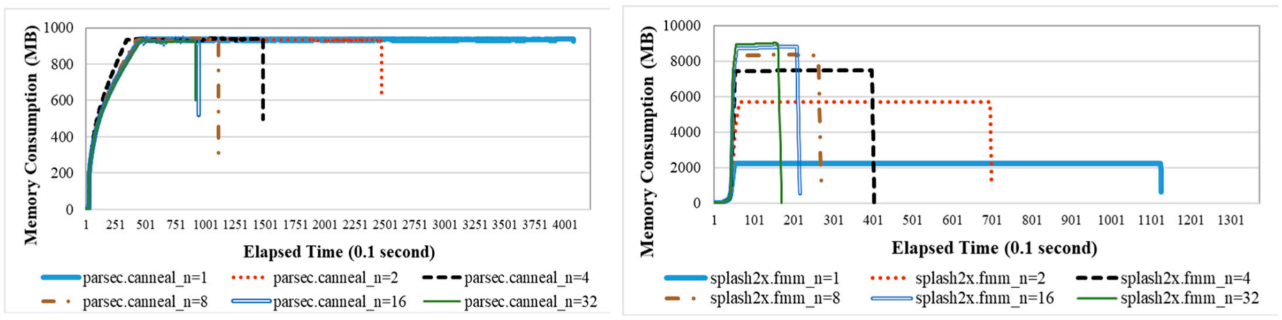


Figure 6. Memory consumption for benchmark programs running with different numbers of threads. (**a**) parsec.canneal; (**b**) splash2x.fmm.

3.2.2. Evaluating Only Those Tasks with Thread Group Leaders on Other Nodes

The observation presented in Section 3.2.1 shows that most of the memory pages allocated to a multithreaded process are on the node where the thread group leader is located. Therefore, ensuring that threads are executed on the same node as the thread group leader involves more local memory access.

For the proposed design, during the task evaluation for inter-node task migration, each task in the runqueue is classified into three types, according to where its thread group leader is currently located. Different decisions are made as follows. If the thread group leader is on the destination node, migrating this task to the destination node may involve more local memory access. Therefore, it is migrated directly instead of evaluating the remaining tasks in the runqueue. Suppose the thread group leader is on the source node. In that case, it is not selected for the migration because migrating it to the destination node may involve more remote memory access after migration. Therefore, only those tasks for which the thread group leaders are currently located on other nodes are evaluated.

Figure 7 illustrates several multithreaded processes running on a 4-node NUMA system. There are four thread groups and 19 tasks. The thread group leaders are denoted as “TGLR.” Because Core#3 is idle, the OS scheduler performs the load balancing mechanism to migrate tasks from the overloaded runqueue (Core#1 on Node#0). The thread group leader for Task#3 is on the source node, so Task#3 is not selected for migration. Similarly, Task#11 is the target task and is migrated immediately because its thread group leader is on the destination node. If more tasks must be migrated to achieve load balancing, the remaining tasks (Task#19, Task#6, Task#16, and Task#9) are evaluated to determine which is best suited to inter-node migration.

3.2.3. Evaluating Only the First Thread Member in a Thread Group

In the source runqueue, only the first member in a thread group is evaluated in the selection procedure. This is because threads in the same thread group share the physical memory pages; their RSS counter values are the same. Figure 8 shows that the Linux kernel represents each thread with a *task_struct* structure, but all threads in the same thread group share the same memory management information (i.e., *mm_struct* structure) and page table. Their RSS counter values (i.e., *mm_rss_stat* structure) are also shared.

For the proposed design, while several tasks in the source runqueue are examined, only the first encountered thread member of a thread group is evaluated; other thread members in the same thread group are not evaluated. Therefore, the identical metric calculations to evaluate the threads of the same thread group in the runqueue are omitted. In the example of Figure 7, for Task#19, Task#6, Task#16, and Task#9, only Task#19 and Task#6 are evaluated.

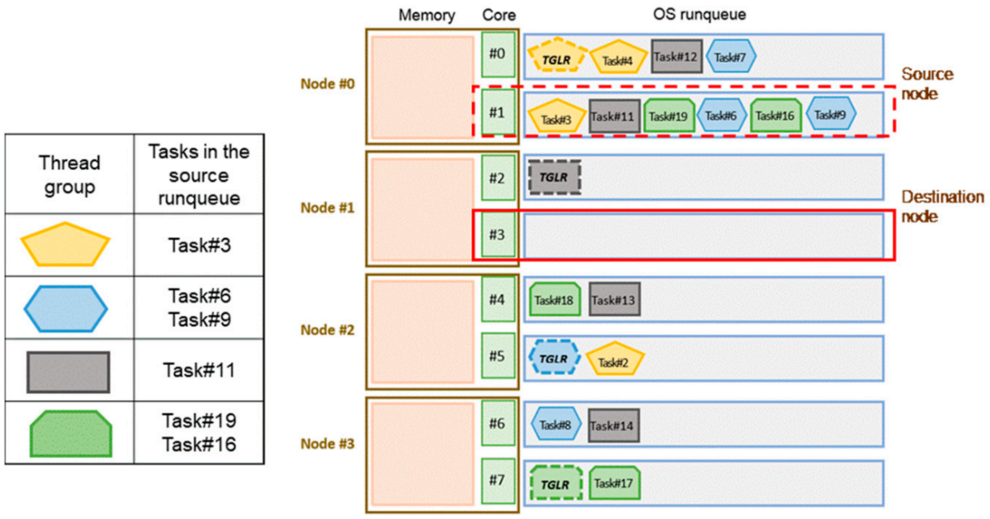


Figure 7. Several multithreaded processes running on a 4-node NUMA system.

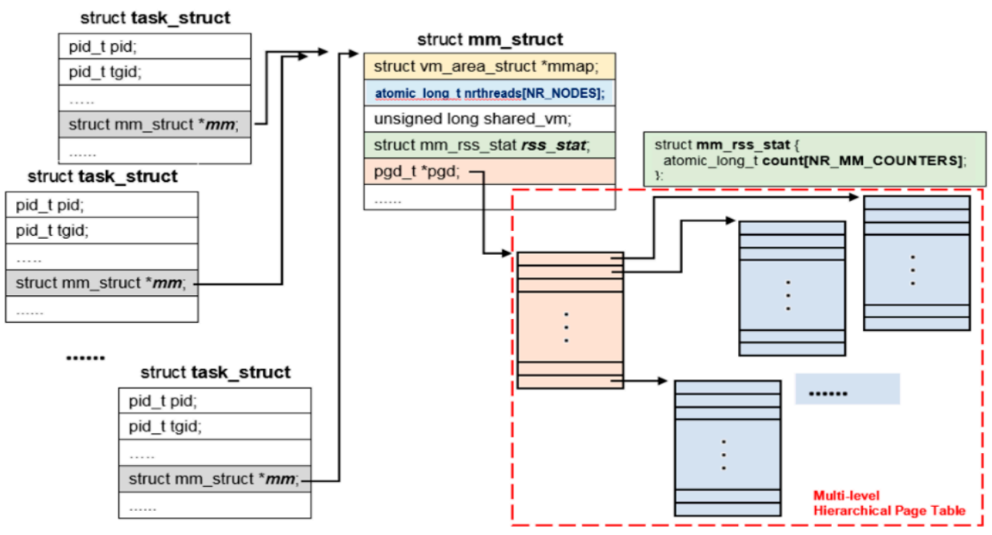


Figure 8. Tasks in a thread group share memory management information.

3.3. *Task Selection Policy with Exclusivity for Multithreaded Applications*

For existing policies, except for the First-Fit policy used in the unmodified Linux kernel, memory-aware policies, such as MB [4] and BCE [4] work with the kMLB mechanism. The Linux kernel must be modified to allow the kMLB mechanism to be used to determine the per-node memory pages per task. When the invoked functions or required data, structures are changed in the newer kernel, these memory-aware policies and the kMLB mechanism also require modification.

This study proposes a new thread-aware policy named Exclusivity (Excl) that does not require the kMLB mechanism. Instead, it considers the exclusivity of thread distribution on nodes for a thread group. The more evenly the threads of a thread group are distributed to nodes, the less beneficial it is for data mapping and thread mapping. Migrating a task across nodes also changes the thread distribution for the thread group. It is better to select a task for which its thread group's threads are distributed more evenly on nodes. The Excl policy selects the task for which the thread group is least exclusive in terms of thread distribution for inter-node migration.

Figure 9 illustrates an example in which 10 tasks belong to two thread groups. Most threads of one thread group are distributed on Node #0, but threads of the other group are evenly distributed on nodes. Migrating Task#10 is more beneficial.

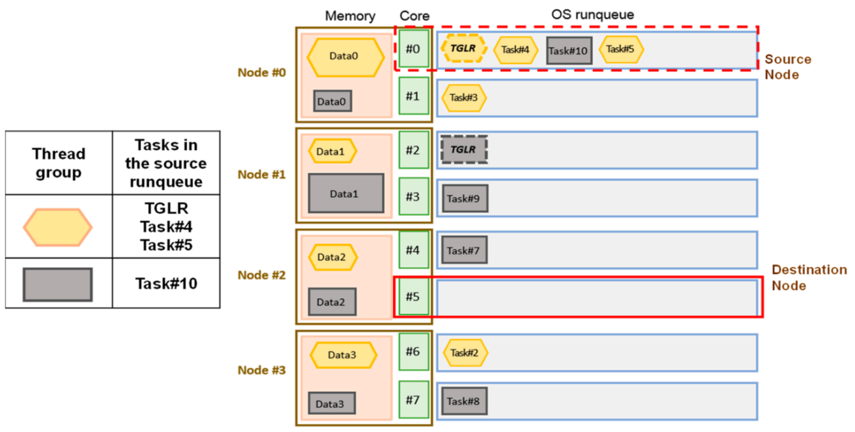


Figure 9. Two thread groups with different levels of exclusivity.

For the proposed policy using the thread distribution of its thread group, for each movable task p in the runqueue, Equation (7) is used to evaluate the exclusivity of this thread group. $thrd\_nr_i$ means the number of threads on NUMA node i, and n is the number of nodes. For tasks in the source core's runqueue, the task for which the thread group has the minimum value for exclusivity is selected. For tasks with equal exclusivity, the first one to be evaluated is the target task:

$$Excl_p = \max_{0 \le i \le n-1} thrd\_nr_i / \sum_{i=0}^{n-1} thrd\_nr_i \tag{7}$$

In Figure 9, threads in hexagon have the value of exclusivity 0.8, and threads in the rectangle have the value of exclusivity 0.4. In Figure 7, Task #11 is selected because it is the least exclusive. The evaluation is listed in Table 3.

Table 3. The evaluation of the Exclusivity policy. (**a**) The evaluation for tasks in Figure 9; (**b**) The evaluation for tasks in Figure 7.

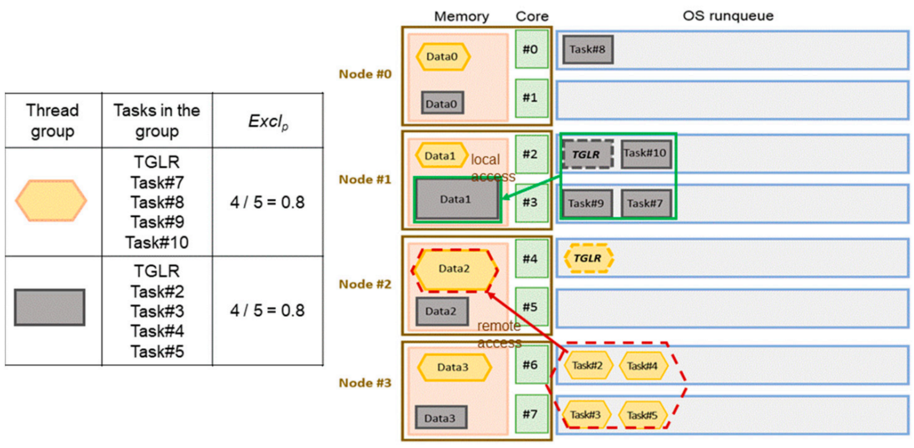
(a)

Thread Group	Thread Group Members in Core#0 Runqueue	$Excl_p$	Selected Task
hexagon ⬡	TGLR, Task#4, Task#5	4/5 = 0.8	Task#10
rectangle ▭	Task#10	2/5 = 0.4	

(b)

Thread Group	Thread Group Members in Core#1 Runqueue	$Excl_p$	Selected Task
pentagon ⬠	Task#3	3/4 = 0.75	Task#11
rectangle ▭	Task#11	2/5 = 0.4	
shape 🟩	Task#19, Task#16	2/5 = 0.4	
hexagon ⬡	Task#6, Task#9	3/5 = 0.6	

However, exclusivity is not the sole criterion for consideration. There is an exceptional case for a multithreaded process for which most of the memory pages are allocated on some nodes, but most threads are on other nodes. In this situation, much remote memory access is necessary. As shown in Figure 10, the thread group in the hexagon and the thread group in the rectangle both have the same value of exclusivity 0.8. However, more remote memory access is necessary for the thread group in the hexagon. Suppose the task for which the thread group is least exclusive is selected. In that case, the highly exclusive thread group may still involve much remote memory access because the thread group leader is on a node different from those where most of the thread group members are located.



Thread group	Tasks in the group	$Excl_p$
⬢	TGLR Task#7 Task#8 Task#9 Task#10	4 / 5 = 0.8
▬	TGLR Task#2 Task#3 Task#4 Task#5	4 / 5 = 0.8

Figure 10. A highly exclusive thread group may involve much remote memory access.

To ensure that threads and the data for a thread group are located on the same node, the proposed policy incorporates the consideration of a thread group leader in the task selection procedure. This makes members of a thread group remain on the same node as the thread group leader. Besides, as described in Section 3.2, most memory pages for a multithreaded process may be allocated by the thread group leader.

Each task in the runqueue is classified into three types according to the node where the thread group leader is currently located. If the thread group leader for a task is on the destination node, this task is not evaluated and is migrated immediately. A task for which its thread group leader is on the source node is not selected for migration. For tasks for which the thread group leader is on other nodes, Equation (7) is used to select the target task for inter-node migration.

4. Performance Evaluation

To measure the performance improvement due to the use of an enhanced inter-node load balancing procedure and the proposed policy for multithreaded applications, the benchmark suite PARSEC 3.0 [3] is used to test systems using different task selection policies and running benchmarks with various numbers of threads. The experiments record the elapsed running time for each test case, several performance counter events, and the elapsed running time for each benchmark program. The results are used to determine the reasons for an increase or decrease in performance.

Section 4.1 details the experimental environment, Section 4.2 describes the experimental design, and Section 4.3 presents the experimental results. Summary and discussions for experimental results are also provided.

4.1. Experimental Environment

The experiments are performed on the NUMA system, Supermicro A+ 4042G-72RF4 [18]. The software and hardware specifications are listed in Table 4. This is a 4-node NUMA system with a constant NUMA factor, and each node is installed with one AMD Opteron 6320 processor [19]. This processor has eight cores, so there is a total of 32 cores in the system. The kMLB [4] mechanism and task selection policies are implemented in the Linux kernel. The Linux *numactl* tool on the system shows that 1.6 times more access is required for remote memory than for local memory.

Table 4. Software and hardware specifications.

System	Supermicro A + 4042G-72RF4 [18] with 4 NUMA Nodes
Processor	4 AMD Opteron 6320 8-core Processor, 2.80 GHz
Motherboard	Supermicro H8QG7-LN4F
Memory controller	2 per node
Interconnect	6.4 GT/s AMD HyperTransport
Cache sizes per processor	L1 Data cache: 16 KB per core L1 Instruction cache: 64 KB per 2-core L2 cache: 2 MB per 2-core L3 cache: 8 MB among all cores per memory controller
Memory	DDR3 1600 16 GB per node (64 GB total)
Operating System	Ubuntu 13.10 Server Edition (Linux kernel 3.11.0-12-generic [20])

The PARSEC 3.0 [6] is a multithreaded benchmark suite that provides a convenient interface for building and running each benchmark program, as shown in Table 2 of Section 3.2.1. The multithreaded configuration "gcc-pthreads" is used to construct most benchmark programs, but the configuration "gcc-openmp" is used to construct the benchmark program parsec.freqmine. The input set "native" is used for performance analysis on real machines to run benchmark programs. The interface allows each benchmark program to be run with the specified number of threads.

4.2. Experimental Design

This study focuses on reducing remote memory access by improving inter-node load balancing. A sufficient number of benchmark programs must be run simultaneously on the experimental system, such that the kernel scheduler migrates tasks between nodes to balance the load when the nodes have an imbalanced load. 29 benchmark programs with a specific number of threads are run simultaneously for each test case. The numbers of threads range from 1, 2, 4, 8, 16, and 32. The elapsed running time for each test case and the elapsed running time for each benchmark program are measured using the performance counter statistics. For each test case, eight to ten runs are performed.

For each run, the system is rebooted to prevent buffer caching. During each run, the performance counter events in Table 5 are also recorded for each benchmark program. These are used to determine the cause of any change in performance: Instructions Per Cycle (IPC), Last Level Cache (LLC), and Miss Per Kilo Instructions (MPKI) to estimate runtime patterns. Other performance data is obtained from *numastat* command-line utility and *vmstat* in the *proc* file system.

Table 5. Collected performance counter events.

Event	Description
Elapsed Time	Used to evaluate the efficiency of running one benchmark program
CPU Cycles	Used to calculate the number of Instructions Per Cycle (IPC)
Instructions	Used to calculate IPC and Last Level Cache (LLC) Miss Per Kilo Instructions (MPKI)
LLC-load-misses	Used to calculate LLC MPKI. LLC-store-misses is not supported.
Page Faults	Number of page faults incurred by threads of one thread group
CPU Migrations	Number of task migrations for one thread group

This study also enhances existing task selection policies as described in Section 2.3 for multithreaded applications, so the experiments use different task selection policies, as shown in Table 6. The First-Fit policy is used in the unmodified Linux kernel. As presented in Section 3.1, the MB [4] policy considers the memory occupied by one task on the destination and source nodes to select a target task for inter-node migration. This policy is suited to multithreaded applications. However, additional overhead is incurred because the policy relies on the information provided by the kMLB mechanism [4].

Table 6. The experimental system running different task selection policies.

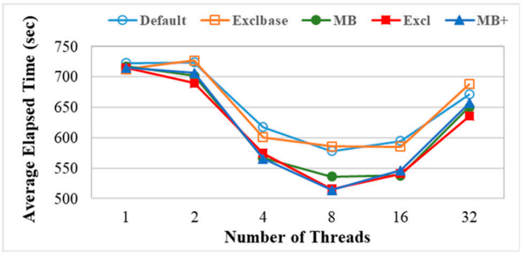
Experimental System	Task Selection Policy	Need kMLB Mechanism	Features
(A)	Default	No	The unmodified Linux kernel with the first-fit policy.
(B)	MB	Yes	The modified Linux kernel with kMLB mechanism and MB policy.
(C)	MB^+	Yes	Based on the MB policy. Enhanced task selection procedure.
(D)	$Excl_{base}$	No	The modified Linux kernel that selects the task for which the thread group has the minimum value of exclusivity.
(E)	Excl	No	Based on the $Excl_{base}$ policy. Enhanced task selection procedure.

To allow faster selection of tasks, we also enhance the MB policy to be MB^+. Each task in the runqueue is evaluated only when its thread group leader is currently not located on the destination or the source nodes. Therefore, only the subset of tasks in the runqueue must be evaluated. Besides, a thread group leader is not migrated across nodes, and a task is migrated directly if its thread group leader is on the destination node.

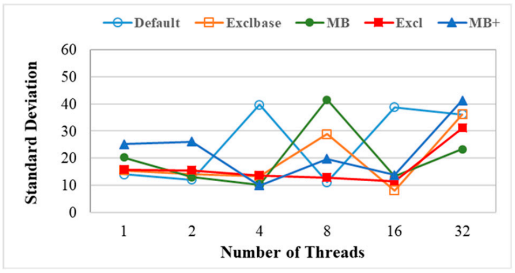
4.3. Experimental Results

4.3.1. Performance Comparison for Varying Numbers of Threads

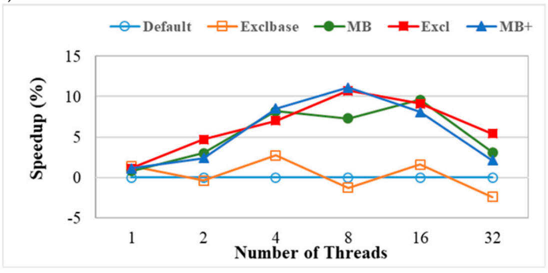
The benchmark programs were run with varying numbers of threads, ranging from 1, 2, 4, 8, 16, and 32. The average elapsed time, the standard deviation, and the ratio of these times to an unmodified Linux kernel are calculated for different task selection policies and the specific number of threads. Figure 11 shows the results.



(**a**)



(**b**)



(**c**)

Figure 11. Experimental results for different numbers of threads. (**a**) Average elapsed time of each test case; (**b**) Standard deviation of each test case; (**c**) The performance increase over the unmodified Linux kernel.

Figure 11a shows that all test cases behave similarly in terms of average elapsed time for different numbers of threads. As the number of threads increases, multithreaded benchmarks run faster since a multi-core system's parallel computing capability increases performance as well. Besides, since more threads wait in the runqueue, an effective task selection policy can select a more suitable one among them for migration. The proposed task selection policies allow more efficient inter-node task migration for the load balancing mechanism on the experimental NUMA system. These perform better than the First-Fit policy that is used in the unmodified Linux kernel.

However, the parallel computing capability of a multi-core system is limited, the elapsed time measured depends on the number of cores in the target system. Because the experimental system has 32 cores, if there are too many threads for a multithreaded benchmark, contention for cores and memory slows down the performance. On the contrary, if the number of threads in a multithreaded benchmark is much smaller, multi-core is not fully utilized. Besides, few or no threads wait in runqueue, then task selection policies are not triggered or used effectively. Therefore, the increase in performance is not so great when the numbers of threads are 1, 2, and 32, as shown in Figure 11c.

4.3.2. The Effect of Enhancing the Task Selection Procedure

The procedure for selecting tasks runs in the critical path of the kernel scheduler, so the target task for the migration must be identified as efficiently as possible. Therefore, this study selects the target task more quickly to allow more efficient inter-node task migration for multithreaded applications. Only the task in the runqueue for which the thread group leader is currently on nodes rather than destination and source nodes is evaluated. No thread group leader is migrated across nodes.

Figure 12 shows the different improvement ratios. The $\text{Excl}_{\text{base}}$ policy achieves a more significant improvement than the MB policy. This enhancement has a different effect on each because the MB policy evaluates each task in the runqueue to select the most beneficial task for migration. For the MB^{+} policy, though this enhancement allows faster task selection by only evaluating the subset of tasks in the runqueue, the selected target task may not result in a greater benefit than the most beneficial task. If there are a few tasks in the runqueue, the saved evaluation costs can be negated by selecting a target task that is not the most suitable.

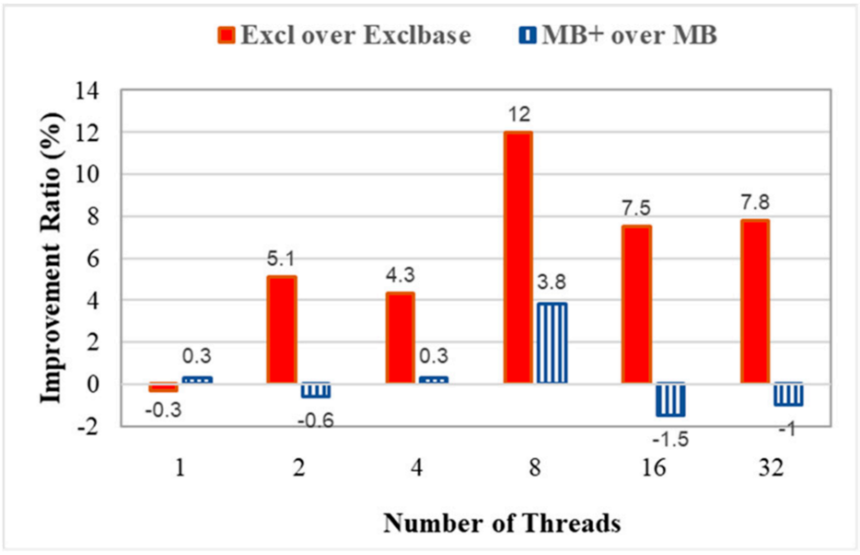


Figure 12. Improvement ratio for enhancing the task selection procedure.

The $\text{Excl}_{\text{base}}$ policy selects the task for which the thread group is least exclusive regarding the thread distribution on nodes. The $\text{Excl}_{\text{base}}$ policy does not consider the examined task's thread group leader, so the selected task can be migrated to the node different from where its thread group leader is currently located. Moving tasks away from its thread group leader can involve more remote memory access after migration. In contrast, the Excl policy incorporates the proposed enhancements for task selection procedure into the Excl$_{base}$ policy. The Excl policy evaluates the task for which the thread group leader is on other nodes and migrates the task to the node the thread group leader is on. Most memory pages are first touched by the thread group leaders, as presented in Section 3.2, so the Excl policy successfully enhances the Excl$_{base}$ policy and improves the performance.

4.3.3. Performance Counter Statistics

We analyze the causes of performance improvements using the measurement data obtained from the performance counter. Since 29 benchmark programs were run with varying numbers of threads, ranging from 1, 2, 4, 8, 16, and 32. They were run on the NUMA server with the Linux kernels using different task selection policies. Lots of figures are obtained from experimental results. From the measurement of the standard deviation of each test case, shown in Figure 11b, the standard deviations for test cases using 4 or 16 threads are relatively small for various task selection policies. Since there are 32 cores in the system, a sufficient number of threads must be run simultaneously on the system. The kernel scheduler then migrates threads between nodes to balance the load when the nodes' loads are imbalanced. As the number of threads increases, more threads wait in the runqueue, and an effective task selection policy can select a more suitable one for migration. Therefore, we observe the runtime patterns for these test cases that use 16 threads. Figure 13 shows the result.

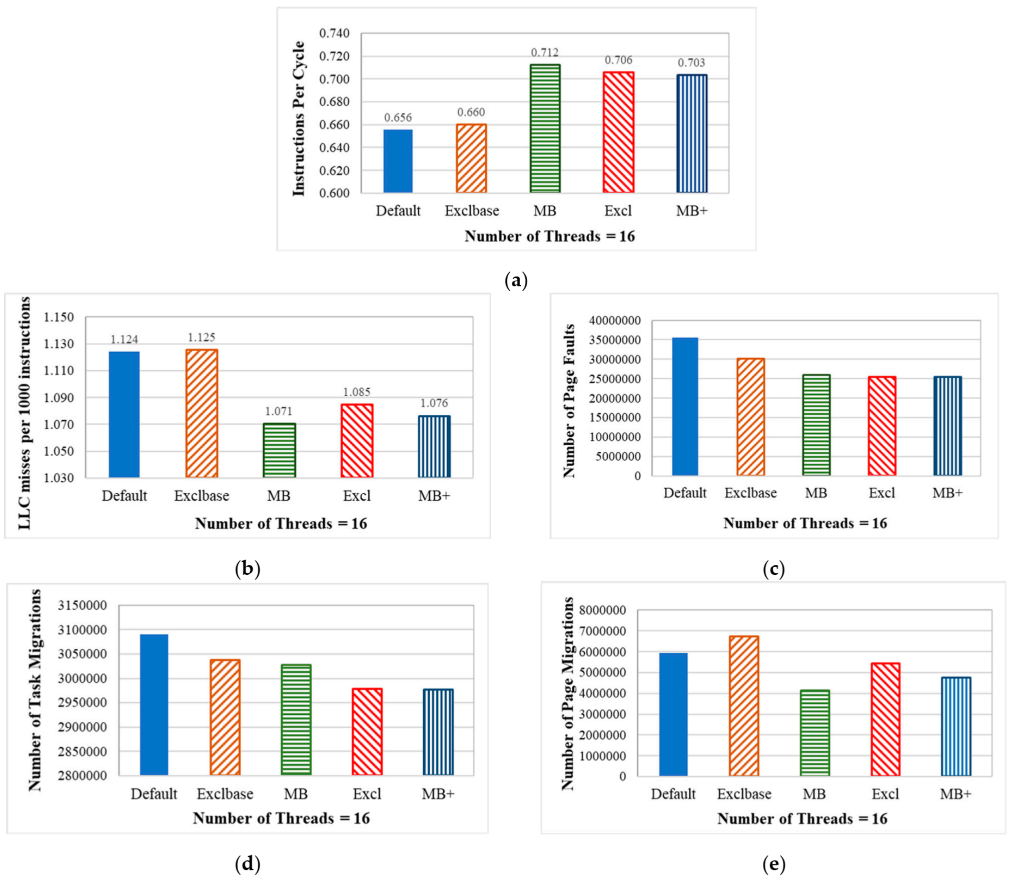


Figure 13. Experimental results for systems using different policies. (**a**) Instructions per CPU cycle; (**b**) LLC misses per 1000 instructions; (**c**) The number of page faults; (**d**) The number of tasks that are migrated; (**e**) The number of pages that are migrated.

Figure 13a,b shows the Instructions Per Cycle (IPC) and the Last Level Cache (LLC) misses per 1000 instructions (LLC MPKI) for systems that use different task selection policies. IPC is used as a reference to evaluate the CPU utilization rate. Cores fetch instructions or data from caches for execution. If the LLC misses, the required instructions or data are then accessed from the main memory. Hence, the LLC MPKI is used as a reference to evaluate the frequency of memory access. The results show that systems using the proposed task selection policies outperform the unmodified Linux kernel, which uses the default First-Fit policy.

We also record the number of page faults, task migrations, and page migrations for each test case, and the results are respectively shown in Figure 13c,d. During the execution of tasks, page faults, task migrations, and page migrations increase the runtime overhead. Systems that use the proposed task selection policies all incur fewer of these operations and have a lower overall runtime overhead than the unmodified Linux kernel, so they perform better.

4.3.4. Performance Results for Various Benchmarks

The performance improvement is measured for each benchmark program. Figure 14 shows the results for each benchmark program running on systems that use different task selection policies for the test cases with 16 threads. We observe that the benchmark programs for which the running time is longer (e.g., parsec.facesim and splash2x.raytrace) obtain more performance gains on systems that use the proposed task selection policies. The benchmark programs (e.g., splash2x.cholesky and parsec.vips) show no performance improvement because their running time is too short. Tasks requiring a longer elapsed time have more chances to be migrated and thus gain more performance improvement.

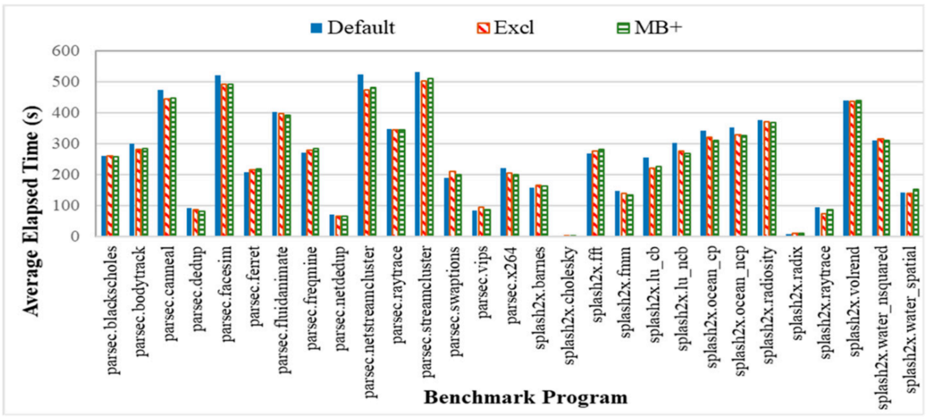
4.3.5. Summary and Discussion

As the analysis presented in Section 3.1, MB is still effective for multithreaded applications. The experimental results demonstrate that MB and MB$^+$ achieve good performance. The experimental results also demonstrate that the Exclusivity policy is competitive. Although its 10.7% performance improvement over the unmodified Linux kernel is still slightly less than the 11.1% performance improvement over the same for MB$^+$ with the kMLB mechanism.

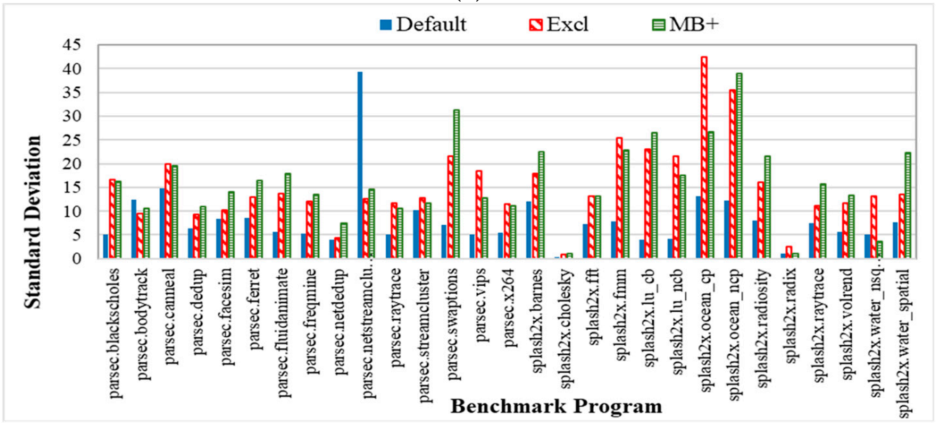
As presented in the study [4], the MB policy works with the kMLB mechanism, and the implementation includes two major works. First, the kernel's memory management routines and exception handling routines are modified to obtain per-task memory usage on each node. Second, the kernel's inter-node load balancing procedure is modified to incorporate the task selection policy. Therefore, to support the kMLB mechanism in the Linux kernel, all kernel operations that update the values of RSS counters and related data structures are modified to track the RSS counters on each node for each process. In detail, seven types of operations are affected and modified, regarding dynamic memory allocation and releases, demand paging, copy-on-write mechanism, swapping, related system calls, and inter-node page migration. However, the latest version of the Linux kernel [20] still does not support the separate counting of memory usage on each node for each process.

In contrast, the Exclusivity policy does not require the kMLB mechanism. Instead, only the kernel's inter-node load balancing procedure is modified to incorporate the task selection policy. Thus, adapting the Exclusivity policy to a newer Linux kernel is less complicated than implementing memory-aware policies with the kMLB mechanism.

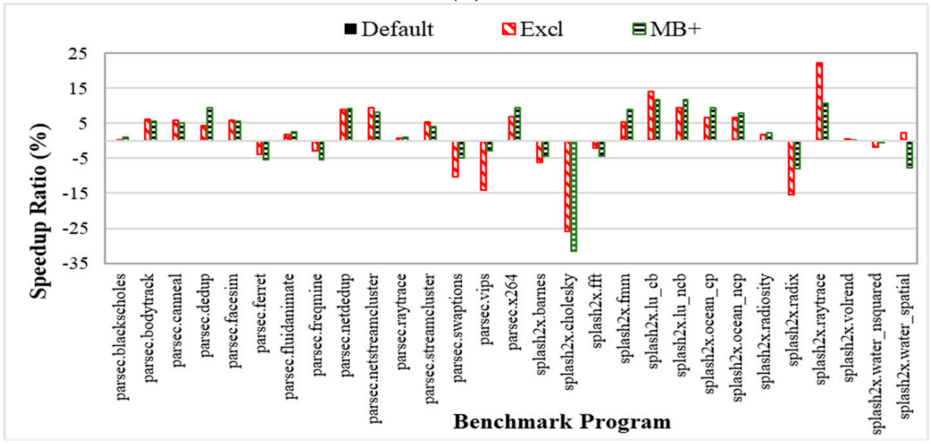
On the other hand, the proposed thread-aware mechanism aims to enhance inter-node load balancing for multithreaded applications on NUMA systems. Therefore, it has some limitations. A sufficient number of threads must be run simultaneously on the system, such that the kernel scheduler then migrates threads between nodes to balance the load when the nodes' loads are imbalanced. As the number of threads increases, more threads are likely to wait in the runqueue, and an effective task selection policy can select a more suitable one among them for migration. In contrast, the default Linux kernel always migrates the first task in the runqueue. Therefore, as shown in the experimental results, its performance is not stable and not good since the first task may not be a good one for migration.



(**a**)



(**b**)



(**c**)

Figure 14. Experimental results for systems running each benchmark program. (**a**) The average elapsed time; (**b**) The standard deviation in the running time; (**c**) The speedup ratio.

However, the parallel computing capability of a multi-core system is limited and depends on the number of cores in the target system. If there are too many threads running, contention for cores and memory resource slows down the performance. On the contrary, if there are too few or no threads wait in runqueue, task selection policies are not triggered or used effectively. The experimental results show that the increase in performance is not significant when the numbers of threads are few or too many.

Regarding the power conservation issue, most modern CPUs support many frequencies. The higher the clock frequency, the more energy is consumed over a unit of time. The longer a thread is running, the more energy is consumed as well.

The Linux kernel supports CPU performance scaling through the CPUFreq (CPU Frequency scaling) subsystem [21]. In our experiments, the Linux kernel uses the default setting of CPU frequency governor "ondemand" [21], which sets the CPU frequency depending on the current usage for all test cases. Since many benchmark programs with many threads are running simultaneously, the system load is high. Therefore, CPUs are set by the Linux kernel to run at the highest frequency during experiments. The experimental results show that benchmark programs run 10.7% faster on the modified Linux kernel with the proposed task selection policy than on the default Linux kernel. Under the same CPU frequency, the shorter the benchmark programs run, the less energy is consumed. The degree of energy-saving needs further experiments.

5. Conclusions

Multi-core systems feature a high throughput, but load imbalance can degrade performance. For NUMA multi-core systems, there is non-uniform memory access, so migrating tasks across nodes to achieve load balancing has a different memory access cost. Therefore, the tasks to be migrated must be selected effectively and efficiently, especially the related operations run in the critical path of the kernel scheduler.

On Linux-based NUMA systems, threads of a multithreaded application share the memory address space and can be scheduled to run on different nodes. Memory pages allocated to them can also be on different nodes. Several studies present specific mechanisms to adjust threads and memory pages on nodes to reduce remote memory access and achieve load balancing. However, strategies using page fault handling to migrate threads or memory pages induce certain overhead. Besides, the kernel scheduler's inter-node task migration can mess up the arrangement. Differently, the kernel-level kMLB [4] mechanism enhances inter-node load balancing for NUMA systems, which tracks the number of memory pages on each node occupied by each task. This memory usage information is then used by memory-aware task selection policies [4] to select the most suitable task for inter-node migration. Despite the required overhead, the kMLB mechanism with task selection policies increases the performance of NUMA systems.

This research studies the memory access for multithreaded processes and proposes a thread-aware kernel mechanism to enhance inter-node load balancing for multithreaded applications on NUMA systems. The proposed Exclusivity policy migrates the task for which its thread group is least exclusive in the thread distribution. A thread group for which tasks are distributed more evenly on different nodes has less impact after task migration. The enhanced task selection procedure does not select the thread group leader for migration to prevent memory pages for one multithreaded process from being scattered on multiple nodes. Besides, only those tasks for which their thread group leaders are on other nodes are evaluated. The proposed policy allows threads of the same thread group to remain on the same node, so performance increases.

This study shows that the Most Benefit (MB) policy is still effective for multithreaded applications, and the proposed Exclusivity policy is competitive with the MB policy. Compared with unmodified Linux, the system that uses the MB$^+$ policy with the kMLB mechanism increases performance by 11.1%. The system that uses the Exclusivity policy, which does not require the kMLB, increases performance by 10.7%. In comparison, it is less complicated to adapt the Exclusivity policy to a newer Linux kernel than to use memory-aware policies with the kMLB mechanism. Moreover, under the same CPU frequency, the shorter the programs run, the less energy is consumed. We plan to adapt our work to a newer Linux kernel and perform experiments on more NUMA systems and energy saving in the future.

Author Contributions: Methodology, M.-L.C. and W.-L.S.; software, W.-L.S.; writing—original draft preparation, M.-L.C. and W.-L.S.; writing—review and editing, M.-L.C.; project administration, M.-L.C.; funding acquisition, M.-L.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in part by Ministry of Science and Technology, Taiwan, grant number 106-2221-E-260-001, 107-2221-E-260-005, 108-2221-E-260-005, and 109-2221-E-260-011.

Acknowledgments: We would like to thank Y. L. Sam for his assistance in performing extensive experiments. Special thanks to the anonymous reviewers for their valuable comments in improving this research.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Bovet, D.P.; Cesati, M. *Understanding the Linux Kernel*, 3rd ed.; O'Reilly Media Inc.: Sebastopol, CA, USA, 2005; ISBN 0596005652.
2. Lameter, C. An Overview of Non-Uniform Memory Access. *Commun. ACM* **2013**, *56*, 59–65. [CrossRef]
3. Chiang, M.L.; Tu, S.W.; Su, W.L.; Lin, C.W. Enhancing Inter-Node Process Migration for Load Balancing on Linux-based NUMA Multicore Systems. In Proceedings of the 10th IEEE International Workshop on Computer Forensics in Software Engineering, Tokyo, Japan, 23–27 July 2018.
4. Chiang, M.L.; Su, W.L.; Tu, S.W.; Lin, Z.W. Memory-Aware Kernel Mechanism and Policies for Improving Inter-Node Load Balancing on NUMA Systems. In *Software: Practice and Experience*; John Wiley & Sons Ltd.: Hoboken, NJ, USA, 2019; Volume 49, pp. 1485–1508.
5. Chen, J.; Banerjee, S.S.; Kalbarczyk, Z.T.; Iyer, R.K. Machine Learning for Load Balancing in the Linux Kernel. In Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems, Tsukuba, Japan, 24–25 August 2020.
6. PARSEC Benchmark Suite. Available online: http://parsec.cs.princeton.edu/ (accessed on 1 June 2021).
7. Diener, M.; Cruz, E.H.M.; Alves, M.A.Z.; Navaux, P.O.A.; Busse, A.; Heiss, H.U. Kernel-Based Thread and Data Mapping for Improved Memory Affinity. In *IEEE Transactions on Parallel and Distributed Systems*; IEEE: Piscataway, NJ, USA, 2016; Volume 27, pp. 1–14.
8. Gennaro, I.D.; Pellegrini, A.; Quaglia, F. OS-based NUMA Optimization: Tackling the Case of Truly Multithread Applications with Non-partitioned Virtual Page Accesses. In Proceedings of the IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, Cartagena, Colombia, 16–19 May 2016; pp. 291–300.
9. Lepers, B.; Quéma, V.; Fedorova, A. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In Proceedings of the 2015 USENIX Annual Technical Conference, Santa Clara, CA, USA, 8–10 July 2015.
10. Li, T.; Baumberger, D.; Koufaty, D.A.; Hahn, S. Efficient Operating System Scheduling for Performance-asymmetric Multi-core Architectures. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Reno, NV, USA, 10–16 November 2007.
11. Terboven, C.; Mey, D.A.A.; Schmidl, D.; Jin, H.; Reichstein, T. Data and Thread Affinity in OpenMP Programs. In Proceedings of the 2008 Workshop on Memory Access on Future Processors, Ischia, Italy, 5 May 2008; pp. 377–384.
12. Unat, D.; Dubey, A.; Hoefler, T.; Shalf, J.; Abraham, M.; Bianco, M.; Chamberlain, B.L.; Cledat, R.; Edwards, H.C.; Fuerlinger, K.; et al. Trends in Data Locality Abstractions for HPC Systems. In *IEEE Transactions on Parallel and Distributed Systems*; IEEE: Piscataway, NJ, USA, 2017; Volume 28, pp. 3007–3020.
13. Goglin, B.; Furmento, N. Memory Migration on Next-touch. In Proceedings of the Linux Symposium, Montreal, QC, Canada, 13–17 July 2009.
14. Goglin, B.; Furmento, N. Enabling High-Performance Memory Migration for Multithreaded Applications on Linux. In Proceedings of the IEEE International Symposium on Parallel & Distributed Processing, Rome, Italy, 23–29 May 2009.
15. Mishra, V.K.; Mehta, D.A. Performance Enhancement of NUMA Multiprocessor Systems with On-Demand Memory Migration. In Proceedings of the 2013 IEEE 3rd International Advance Computing Conference, Ghaziabad, India, 22–23 February 2013; pp. 40–43.
16. Marchetti, M.; Kontothanassis, L.; Bianchini, R.; Scott, M. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-memory Systems. In Proceedings of the 9th International Parallel Processing Symposium, Santa Barbara, CA, USA, 25–28 April 1995.
17. Schermerhorn, L.T. Automatic Page Migration for Linux. In Proceedings of the Linux Symposium, Sydney, Australia, 15–20 January 2007.
18. Supermicro AS 4042G-72RF4. Available online: https://www.supermicro.com/Aplus/system/Tower/4042/AS-4042G-72RF4.cfm (accessed on 1 June 2021).
19. AMD Opteron 6320 Processor. Available online: https://www.amd.com/en/products/cpu/6320 (accessed on 1 June 2021).
20. The Linux Kernel Archives. Available online: https://www.kernel.org/pub/linux/kernel/v3.0/ (accessed on 1 June 2021).
21. Linux CPUFreq Governors—Information for Users and Developers. Available online: https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt (accessed on 4 July 2021).