A Novel Memory-Aware CPU Allocation Policy for Multicore NUMA Architecture

Dongwoo Kang[1], Heekwon Park[2], and Jongmoo Choi[3]

[1] School of Computer Science Engineering, Dankook University, Yongin, Korea, 448-701, rediori@dankook.ac.kr
[2] School of Computer Science Engineering, Dankook University, Yongin, Korea, 448-701, parkhk81@dankook.ac.kr
[3] School of Computer Science Engineering, Dankook University, Yongin, Korea, 448-701, choijm@dankook.ac.kr, Corresponding Author

Summary. Recent computer systems, ranging from mobile systems to servers, are employing multicore processors such as Intel's Xeon, AMD's Opteron and ARM's Cortex-A9. Also, to reduce bus contention among multiple cores and DRAMs, they adopt the NUMA (Non-Uniform Memory Access) memory organization, wherein each core has direct path to its own local memory, leading to access local memory faster than remote memory. These trends of computer architecture trigger to rethink the internal structures and policies of today's operating system. In this paper, we design a new memory-aware CPU allocation policy for multicore NUMA architectures that has the following three features. First, it makes a CPU allocation decision based on not only CPU load but also memory load, which enables to decrease the possibility of referencing remote memory. Second, it applies different weight on CPU load and memory load hierarchically and adaptively according to the types of CPU allocation requests. Finally, it utilizes the characteristics of processes such as CPU intensity and memory intensity to accurately estimate the CPU and memory load of each core. Real implementation based experimental results have shown that the proposed memory-aware CPU allocation policy can actually enhance the execution time of applications, compared with the traditional Linux CPU allocation policy.

1 INTRODUCTION

One of the recent trends of computer architecture is employing multiple cores. Intel has already demonstrated a 32nm-based six-core processor, named the *Core i7* processor, and has a plan to provide an eight and more cores processor [1]. AMD also has released new the *Opteron* architecture with 8 and 12 cores on each processor [2]. Even in the embedded system domains, ARM has developed the *Cortex-A9* multicore processor, consisting of 4 ARM CPUs with low power-consumption features [3]. When we equip two or more processors in computer systems, we can easily configure tens or hundreds of CPUs as our computing resources.

As the number of cores increases, the possibility of bus conflicts between multiple cores and DRAMs also increases. To mitigate the bus conflicts, NUMA (Non Uniform Memory Access) memory organizations are popularly adopted in multicore systems, such as *Intel's Nehalem QPI(QuickPath Interconnect)* [1] and AMD's *Hypertransport* technologies [2]. In these organizations, the access latency of remote memory is slower than that of local memory. Therefore, placement of processes and data among local and remote memory become critical for obtaining high performance like as in GPGPU (General Purpose Graphical Processing Unit) architecture [4].

These architectural trends require to revisit the internal structures and policies of today's operating systems [5] [6]. In this paper, we investigate a novel memory-aware CPU allocation policy for multicore NUMA architectures. To begin with, we have made two observations on real Intel Xeon 8 cores (2 processors) NUMA systems. The first observation is that the latency differences between local and remote memory are quite large, which gives a negative effect on the execution time of a process when it references remote memory frequently. The second observation is that the performance degradation of memory-bound processes is much larger than that of CPU-bound processes when they are allocated on the same core.

From the observations, we design a new CPU allocation policy that has the following three features. The first one is that, it makes a CPU allocation decision based on not only CPU load but also memory load so that it can maximize the CPU and memory load balancing simultaneously. The memory load balancing gives an opportunity to diminish the number of accesses for remote memory. The second feature is applying different weight on CPU load and memory load hierarchically and adaptively according to the types of CPU allocation requests. In particular, it gives more weight on CPU load for the CPU allocation request at the process creation time (i.e.. sys_fork( )), while more weight on memory load for that at the process loading time (i.e.. sys_execve( )). Finally, it utilizes characteristics of processes such as CPU intensity and memory intensity to estimate accurately CPU load and memory load of each core.

The proposed CPU allocation policy has been implemented on Linux kernel version 2.6.32, based on the hardware platforms consisting of Intel Xeon X5570 8 cores (2 processors), 32GB DDR3 DRAM, 4TB SAS Disk, and peripherals. The 32GB DRAM is divided into two memory units where the first processor uses one unit as local memory and the other as remote memory, and vice versa. Experimental results with three benchmarks have shown that the proposed policy can actually enhance the execution time of each benchmark, compared with the traditional Linux CPU allocation policy.

The rest of this paper is organized as follows. In the next section, we discuss the motivation of this work. Then, we explain the design issues and implementation details in Section 3. In Section 4, we present the performance evaluation results and finally, we conclude this paper in Section 5.

2 MOTIVATION

In this section, we discuss two observations conducted on the Intel Xeon 8 cores (2 processors) NUMA system that motivate our research. We first explain the structure of the experimental system briefly, and then elaborate how it affects the performance of Linux kernel.

Fig 1 depicts the structure of the Intel Xeon multicore NUMA system considered in this paper. It consists of 2 processors, each of which in turn consists of 4 cores (in this paper, we use the terms of CPU and core interchangeably). According to the interrelation between cores and DRAMs, multicore system can be classified into two categories, one is the UMA(Uniform Memory Access) organization and the other is the NUMA(Non-Uniform Memory Access) organization.

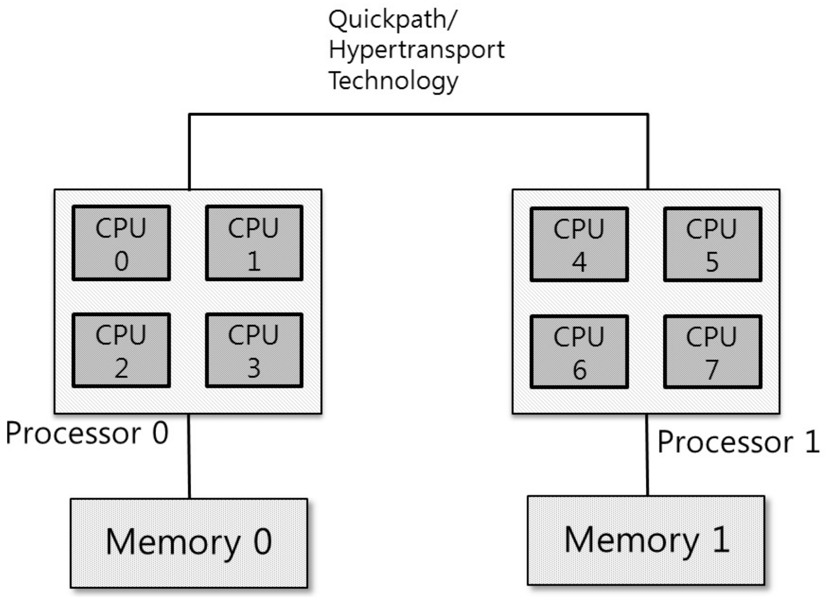

Fig. 1. Multicore NUMA architecture

In UMA, all cores can access any memory address at the same latency. On the contrary, in NUMA, some memory banks are connected directly to a processor or processors, while others are connected through the interconnection technology such as Intel's QuickPath Interconnect and AMD's HyperTransport. In the example of Fig 1, processor 0 can access memory 0 directly, while accessing memory 1 through the interconnection technology. Directly accessible memory is defined as local memory while other memory is defined as remote memory. In Fig 1, memory 0 and memory 1 become local memory and remote memory, respectively, for a processor 0 and vice versa, for a processor 1. Note that although we can equip more processors and DRAMs and construct more complex interconnected multicore NUMA systems, the concepts of local and remote memory and our suggestions of this paper are still valid in such systems.

The first observation is measuring the latency differences between local and remote memory quantitatively and analyzing the performance effects of multicore NUMA architecture on operating systems. For this purpose, we have ported the Linux kernel version 2.6.32 on our experimental system and have executed a synthetic application that accesses data with the size of 128MB sequentially. Then, we have measured the total elapsed time of the application when data is allocated on local memory and remote memory, respectively, as shown in Fig 2.

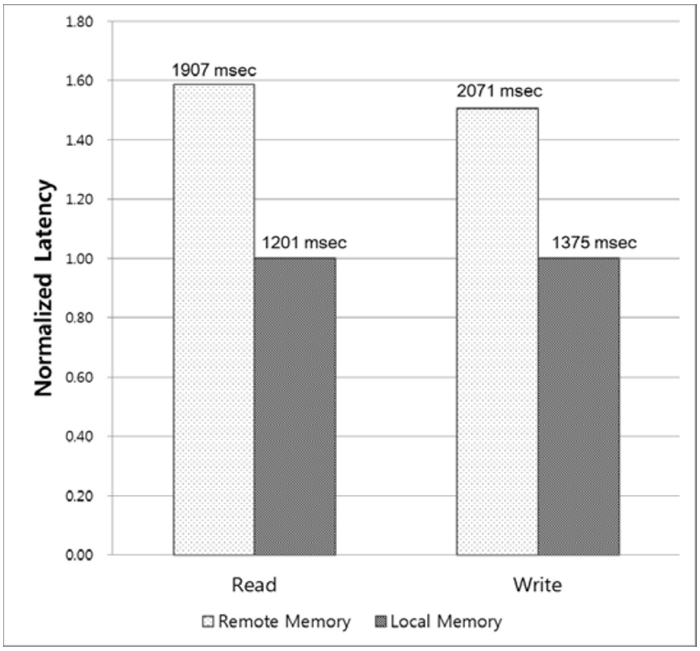

Fig. 2. Latency differences between local memory and remote memory

From Fig 2, we can observe that the latency of remote memory is quite slower than that of local memory. On average, read accesses for remote memory takes 1.6 times longer than those for local memory, while write accesses takes 1.5 times longer. The results reveal some performance aspects of operating system on multicore NUMA systems. One aspect is that a process might have considerable performance degradation if operating system allocates page frames for the process from remote memory. The other aspect is that a naive

CPU allocation policy and/or process migration policy across multiple processors for CPU load balancing may cause a bunch of remote memory references, causing nontrivial performance deterioration.

Our second observation is the scalability of multiple processes on multicore NUMA systems. For this experiment, we have built two different types of synthetic applications. One is a CPU-bound application that executes several mathematical operations intensively and the other is a memory-bound application that allocates 32MB memory space and executes memory read/write operations repeatedly. Then, we have created various numbers of processes where each process executes the CPU-bound or memory-bound application independently and have measured the average runtime of processes, as shown in Fig 3. Note that we carefully configure the runtime of each application as 1 second to plot the graph of Fig 3 more intuitively. Also note that, in this experiment, memory space requested by each process is allocated only from its local memory so that we can focus on scalability analysis purely without the effects of remote memory references.

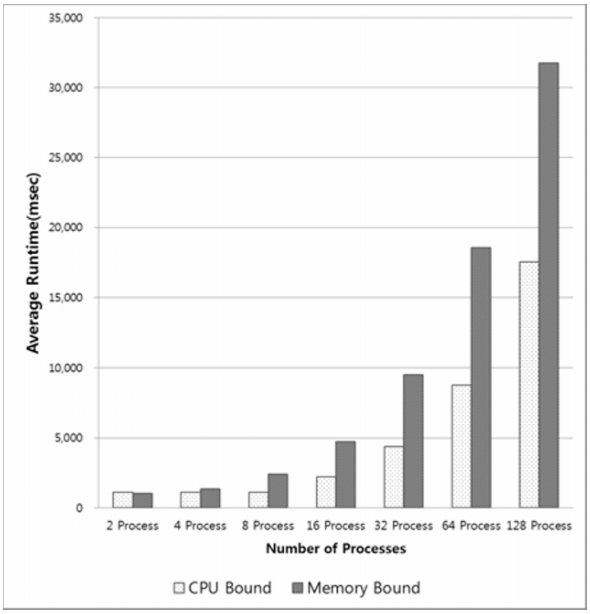

Fig. 3. Scalability of CPU-bound and Memory-bound processes

Fig 3 illustrates that CPU-bound processes show better scalability than memory-bound ones. Let us discuss the results of the CPU-bound processes first. When the number of processes increases from 2 to 8, the average runtime of each process is constantly 1 second, the same as the runtime of the CPU-bound application. This is because there are 8 cores in the experimental system, as depicted in Fig 1, and each core can execute each process concurrently. When the number of processes becomes 16, the average runtime becomes 2 seconds since each core needs to run two processes with an interleaving fashion. These trends continue as shown in Fig 3. As the number enlarges twice, the time also increases twice.

On the contrary, memory-bound processes show a relatively poor scalability. As we expect, the addition of processes leads to increase the average runtime of processes. However, the increasing ratio of memory-bound processes is larger than that of CPU-bound processes, even though all processes access local memory only. Observations from Fig 2 imply that the increasing ratio becomes much larger if some processes begin to access remote memory. It indicates that, in multicore NUMA architecture, memory contention might be more critical than CPU contention to achive good performance. In other words, we need to consider carefully not only CPU load balancing but also memory load balancing for designing efficient CPU allocation policies.

3 DESIGN AND IMPLEMENTATION

Based on the observations discussed in Section 2, we design a new memory-aware CPU allocation policy. The key feature of the proposed policy is that it makes a CPU allocation decision by using both CPU load and memory load. Traditionally, the conventional CPU allocation policies make use of CPU load only since they are mainly focusing on the UMA memory organization [7] [8]. However, our policy exploits memory load together to reduce the number of remote memory references.

Also, there are two additional features in our policy. First, to reflect the characteristics of processes such as CPU-bound and memory-bound, we define CPU intensity and memory intensity of a process and apply them to estimate CPU load and memory load of each core. Secondly, for CPU allocation, we give a different weight on CPU load and memory load hierarchically and adaptively according to the allocation request type.

3.1 Estimate CPU load and Memory load

Traditionally, operating systems make their CPU allocation decisions based on CPU load only and one of the commonly used techniques for estimating CPU load of a core is counting the number of processes in the run queue assigned to the core. However, for better CPU allocations in multicore NUMA systems, it needs to contemplate the characteristics of processes as discussed in Fig 3. To reflect this requirements, we introduce two new attributes of a process, which are CPU intensity and memory intensity.

CPU intensity is a concept to represent how much actively a process utilizes CPU resources. In this paper, we define it as the ratio between the possible time slice allotted to a process and the actual used time slice. Formally, it can be expressed as follows:

$$C_i(k) = \frac{ConsumedTS(k)}{AllocatedTS(k)} \tag{1}$$

where $C_i(k)$ is the CPU intensity of a process *k*, and *AllocatedTS(k)* and *ConsumedTS(k)* are the allocated time slice allowed to consume and the actually consumed time slice at the last execution, respectively. For instance, when a process, that has the allocated time slice of 100 milliseconds , actually consumed 40 milliseconds at the last scheduled time, the CPU intensity of the process becomes 0.4.

Similarly, memory intensity is defined as how much working set a process has. It can be expressed as follows:

$$M_i(k) = \frac{UsedPF(k)}{ProportionalSharePF(k)} \tag{2}$$

where $M_i(k)$ is the memory intensity of a process k and *UsedPF(k)* is the number of page frames currently used by the process. Also, the *ProportionalSharePF* is the average number of page frames possibly allocated to a process if all processes use page frames fairly. It can be calculated by dividing the total number of page frames in a system with the current number of processes. As an example, if there are 1000 page frames and 5 processes in a system and a process is currently using 40 page frames, then the memory intensity of the process is 0.2.

The introduction of CPU intensity and memory intensity enables to estimate CPU load and memory load of a core through the following two formulas:

$$C_{load}(i) = \sum_{k=0}^{N} C_i(k) \tag{3}$$

$$M_{load}(i) = \sum_{k=0}^{N} M_i(k) \tag{4}$$

where $C_{load}(i)$ and $M_{load}(i)$ are CPU load and memory load of a core i, respectively, and k (range from 0 to N) are the processes assigned to the core.

For example, let us assume that a system has two cores and each core has three processes. Also assume that each process is described with the two parameters of (allocated time slice, consumed time slice), and the three processes on the first core are (100, 20), (100, 20) and (100, 50), while the other three processes on the second core are (100, 20), (100, 10), (200, 60). Then, CPU load of the first core is 0.9, while that of the second core is 0.6. From these estimates, we can recognize that the first core has higher CPU load than the second core. Note that by estimating from the number of processes or from the total execution times of processes, we can not differentiate CPU loads among the cores.

As an another example, assume that the system described at the previous paragraph has total 1200 page frames and the three processes of the first core currently use 40, 80, 40 page frames while the other three processes of the second core use 20, 160, 40 page frames, respectively. Then, memory intensity of the first core is 0.8 while that of the second core is 1.1. These estimates enable to identify how much memory space a core currently uses and how the potentiality of a core to allocate page frames from remote memory is. Note that the working set size of a process can be changed during the execution. However, decent operating systems can adapt the changes well using a variety of page allocation and replacement techniques [7] and we conjecture that measuring the current resident page frames of a process is sufficient to estimate memory intensity for our memory-aware CPU allocation policy.

3.2 Hierarchical and Adaptive CPU allocation

Now the question is how to make use of the estimated CPU load and memory load for CPU allocation. One simple way is just adding the two loads and choosing a core that has the smallest sum of the loads. However, since CPU load and memory load represent distinct characteristics of a process and, in general, a system has different capacity of CPU and memory resources, it needs to give a different weight on the two loads. For this purpose, we devise a new formula that is expressed as follow:

$$CM_{load}(i) = \alpha \cdot C_{load}(i) + (1 - \alpha) \cdot M_{load}(i) \tag{5}$$

where $CM_{load}(i)$ is the weighted combination of CPU load and memory load of a core i and α is a control parameter ranging from 0 to 1. The role of the control parameter α is determining the weight on CPU load and memory load. Specifically, as α approaches 1, the policy gives more weight on CPU load and, eventually when α is equal to 1, it considers CPU load only. On the contrary, as α goes to 0, the policy gives more weight on memory load and, finally it reflects memory load purely when α is equal to 0.

There are two feasible approaches to decide the appropriate value of the parameter α. The first one is choosing the value that can minimize the average response time of all processes on a system. It's kind of an optimization problem, finding the best solution with the consideration of given conditions such as number of processes, characteristics of each process, and CPU and memory capacities. The second approach is exploiting operating system heuristics such as scheduling level and request types that can provide useful hints to determine the value of α. In this paper, since the second approach can be practically implemented in real operating systems and executed on-line without considerable overheads, we select the second approach and leave the first one as a future work.

To explore useful operating system heuristics, we first analyze the scheduling behaviors of Linux kernel. Fig 4 presents the scheduling mechanism used

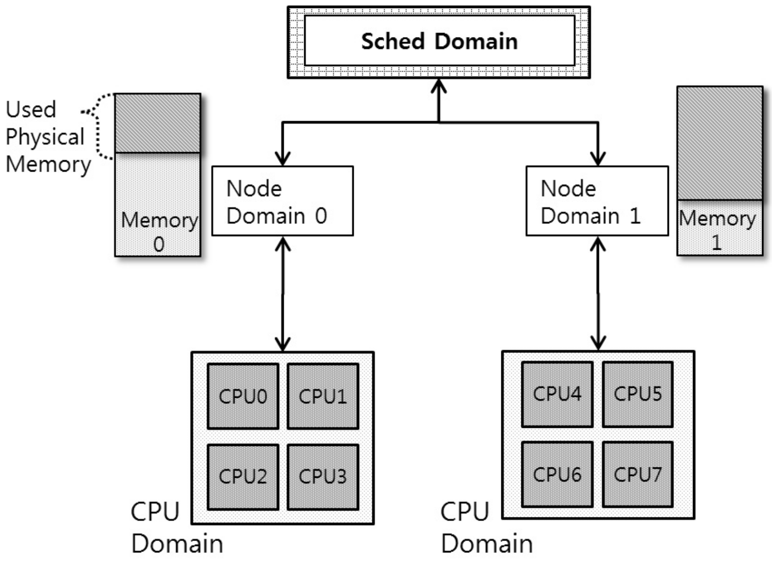

Fig. 4. Scheduling infrastructure in Linux Kernel

in Linux kernel, where CPU allocation is conducted hierarchically at two levels, one is a node domain level and the other is a CPU domain level. Linux kernel uses a data structure, called *sched_domain*, which arranges CPUs in a hierarchy depending on physical hardware organizations [10]. In the NUMA organization, the sched_domain consists of several *node domains*, and each node domain, in turn, consists of a set of CPUs that have the same viewpoint about local and remote memory. In other words, we can regard the node domain as the processor depicted in Fig 1.

The traditional CPU allocation policy in Linux kernel chooses a CPU using the following three steps: 1) trying to choose the same CPU, 2) trying to choose a CPU from the same node, 3) selecting the most idlest CPU from all nodes. Before elaborating these steps in detail, we define some terms first. The *original CPU* of a process is defined as the CPU where the process has run on at the previous execution. Similarly, the *original node* of a process is defined as the node that contains the original CPU of the process. For instance, assume that a process was executed on CPU 2, then the original CPU and node of the process are CPU 2 and node 0, respectively in Fig 4. The final term is the weight of a process in Linux kernel. In the Linux kernel version 2.6.23 and later, every process has its weight value depending on its priority. As an example, if a process has the default priority of 120, Linux kernel assigns 1024 into the process as the weight of the process. Also, it assigns 1.25 times larger weight value into a process with the priority of 119 higher value, and vice versa [11].

Now, let us elaborate the three steps of the traditional CPU allocation policy used in Linux kernel. At the first step, it tries to allocate the original CPU to achieve CPU cache affinity and to minimize process migration overheads. Specifically, it adds up the weights of all processes on the original CPU and, if the value is lower than a threshold value, it allocates the CPU to the process. Otherwise, Linux kernel regards the CPU as too congested and goes into the second step. The default value of the threshold is set as 1024 in Linux kernel. At the second step, it tries to allocate a CPU from the original node. In specific, it selects a CPU with the least sum of weights of processes from the original node. When the value is lower than the threshold value, the CPU is allocated to the process. The reason of this trial is to execute a process at the same node, which enable to reduce the number of remote memory accessing.

If it fails to select a CPU at the second step again, it moves into the final step. Here, it chooses a node that has the least sum of weights of processes among nodes. Then, it selects a CPU that has, again, the least weighted number of processes from the chosen node. In Linux, the *find_idlest_group()* and the *find_idlest_cpu()* functions take care of the choice of a node and a CPU, respectively.

From the analysis of the CPU allocation mechanism in Linux kernel, we notice that CPU allocation is carried out hierarchically at the two levels, one is the node domain level and the other is the CPU domain level. Also, we find out that the two levels have quite antithetic requirements. At the node domain level, memory load might be more important for CPU allocation. This is because recent developed multicore NUMA systems provide almost identical CPU capabilities among nodes, while they show considerably differences of memory latencies among local and remote memory. On the contrary, at the CPU domain level, CPU allocation based on memory load does not give any performance differences since all CPUs in a node have the same memory configuration.

These findings trigger us to design a new CPU allocation policy that gives more weight on memory load at the node domain level and CPU load at the CPU domain level. These controls are possible using the formula 5) with various values of α. In this paper, we set α as 0 at the node domain level while setting it as 1 at the CPU domain level. Setting α as 1 at the CPU domain level is reasonable since memory load does not bring any differences at this level. However, setting α as 0 at the node domain level could be controversial and there might be more appropriate value. There are two reasons for our setting. One is that as the number of cores increases, CPU becomes comparatively abundant resources. The other, and more substantial, reason is that, after the node domain level, we still have another chance to reflect CPU load at the CPU domain. Analyzing the effects of α with various values is left as a future work.

In addition to the hierarchical adaptation, we apply CPU load and memory load differently according to the types of CPU allocation requests. In Linux kernel, CPU allocation is requested mainly by the three sources; pro-

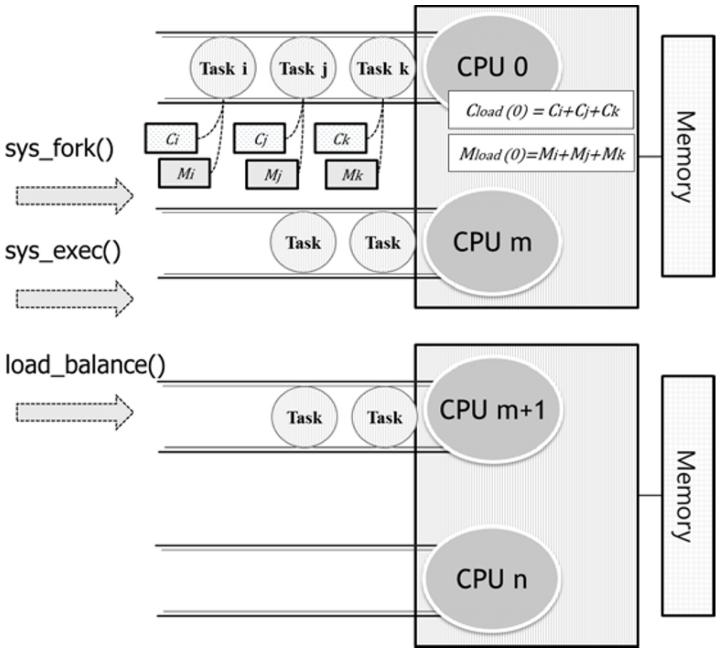

Fig. 5. Linux Kernel internals for CPU allocation

cess creation (sys_fork), process image loading (sys_execve), and load balancing (load_balance), as shown in Fig 5. We carefully examine the behaviors of the three request types and design appropriate CPU allocation procedures for each type, as described pseudo code in Algorithm7. The process creation request, denoted as sys_fork in Algorithm 7, creates kernel data structures such as *task, file* and *mm* structures, and makes the new child process and the original parent process share memory images (text segment is actually shared, while data segment is shared through the copy-on-write mechanism). Hence, when the child and parent processes are assigned into different nodes, one of them needs to access remote memory frequently, which incurs performance degradation. To overcome this problem, we need to assign the child process into the same node of the parent process. Hence, we choose statically the node of a parent process as a candidate node. Then, we select a CPU from the candidate node with $\alpha = 1$(consider CPU load only).

On the other hand, the process loading request discards the existing memory images and brings in new program images from disks. Since the discarding makes it free to assign the process into whether local or remote memory, it's a good chance to enhance memory load balancing. Therefore, we first select a node with α= 0 (consider memory load only). Then, we choose a CPU from the selected node with α= 1.

Algorithm 7 CPU Allocation algorithm

```
  $NODE \leftarrow \{node_1, node_2, \ldots, node_i\}$                ▷ The set of all nodes
  $CPU[i] \leftarrow \{cpu_1, cpu_2, \ldots, cpu_k\}$          ▷ The set of all CPUs in node of $i$

Require: $node\_id \in NODE$
Require: $x \in CPU[node\_id]$                          ▷ x is set of CPUs in node of $i$
  procedure SELECT_CPU($node\_id$,$\alpha$)    ▷ Select a CPU in the node_id domain that has
  the smallest $CM_{load}$ with the parameter $\alpha$
      $candidate\_cpu \leftarrow 0$, $smallest\_cm_{load} \leftarrow \infty$
      while $all\ x$ do                                   ▷ We have the answer if r is 0
          if $smallest\_cm_{load} > CM_{load}(x)$ then
              $candidate\_cpu \leftarrow x$
              $smallest\_cm_{load} \leftarrow CM_{load}(x)$
          else
          end if
      end while
      return $candidate\_cpu$                               ▷ The idlest cpu number
  end procedure

Require: $i \in NODE$
Require: $j \in CPU[i]$
  procedure SELECT_NODE($\alpha$)    ▷ Select a node that has the smallest $CM_{load}$ with the
  parameter $\alpha$
      $candidate\_node \leftarrow 0$, $smallest\_cm_{load} \leftarrow \infty$
      while $all\ i$ do
          $weights \leftarrow 0$
          while $all\ j$ do
              $weights \leftarrow weights + CM_{load}(j)$
              if $weights < smallest\_cm_{load}$ then
                  $candidate\_node \leftarrow i$
                  $smallest\_cm_{load} \leftarrow weights$
              end if
          end while
      end while
      return $candidate\_node$
  end procedure

  procedure SELECT_TASK_RQ($request\_type$)    ▷ CPU allocation main function, applying
  CPU and memory loads adaptively and hierarchically
      $candidate\_node \leftarrow 0$
      if $request\_type = sys\_fork$ then
          $candidate\_node \leftarrow parant\_node$
          $candidate\_cpu \leftarrow Select\_CPU(candidate\_node, 1)$
      else if $request\_type = sys\_execve$ then
          $candidate\_node \leftarrow Select\_Node(0)$
          $candidate\_cpu \leftarrow Select\_CPU(candidate\_node, 1)$
      else $request\_type = load\_balance$
          if $process\_remote\_memory > process\_local\_memory$ then
              $candidate\_cpu \leftarrow Select\_task\_rq(sys\_execve)$
          else
              $candidate\_cpu \leftarrow Select\_task\_rq(sys\_fork)$
          end if
      end if
      return $candidate\_cpu$
  end procedure
```

The final request type is load balancing. The load balancing gives a positive performance effects by migrating processes from congested CPUs into idle CPUs. However, especially in NUMA systems, it may cause performance degradations due to the increasing of remote memory references when a migration is conducted across nodes. So, we need to carefully consider the tradeoffs. In this paper, if a candidate process for load balancing has more page frames in local memory, we handle it like the process creation request so that the process is allocated into the same node. Otherwise, we handle it like the process loading request so that the process can be allocated into a node with the least memory load.

We have implemented the proposed memory-aware CPU allocation policy in Linux kernel version 2.6.32. Table 1 summarizes the modified files and their descriptions.

Table 1. Implementation Summary

Modified File	Descriptions
sched.c	1) Add two new variables for manipulating CPU_{load} and $memory_{load}$ in the run queue of each CPU (struct cfs_rq) 2) Add two new variables for manipulating C_i and M_i of a process (struct task_struct) 3) Measure the allocated time slice and the consumed time slice (sched_slice(), sum_exec_runtime and prev_sum_exec_runtime in struct sched_entity)
sched_fair.c	1) Implement the proposed memory-aware CPU allocation functions described in Algorithm 7. 2) Update C_i and CPU_{load} and M_i and M_{load} when a process put in or get out of the run queue of a CPU (put_prev_task(), pick_next_task())
mm.h	1) Measure the used page frames and M_i of a process. (_file_rss and _anon_rss in struct mm_struct)

For implementing the proposed policy, we need to measure several process information such as the allocated time slice, the consumed time slice, and the resident page frames of a process. Fortunately, such information can be obtained from the existing data structures and functions, which allows to implement our policy without considerable overheads to gather information and without impairing the portability of Linux kernel. Also, we make use of several lightweight CPU load and memory load monitoring techniques such as hashing and pre-calculation to minimize runtime overheads.

One concern we have in the implementation of the propose policy is the overhead for communication between user processes and Linux kernel. For example, assume that Linux kernel runs in a node 0 of Fig 4. Then, a process that runs in a node 1 might have some performance degradation due to the communication between kernel and the process such as system calls and data copies. To mitigate this problem, we consider that the policy first tries to allocate CPUs from the node where Linux kernel runs on, if the number of processes on the node is less than a controllable threshold value. However, this choice does not give any performance benefits. Sensitivity analysis has shown that Linux kernel has already provided the replications of kernel images both on local and remote memory to overcome the problem [11]. As a result, we omit this consideration from the current implementation.

4 PERFORMANCE EVALUATION

We have evaluated the performance of the proposed memory-aware CPU allocation policy on a real system. The experimental system consists of Intel Xeon X5570 8 cores (2 processors), 32GB DDR3 DRAM, 4TB SAS Disk, and peripherals. On this hardware platform, we have executed three benchmarks, named nbench, STREAM, and lmbench . The nbench is designed to expose the capabilities of CPU and memory in a system [12], while the STREAM is used for testing sustainable memory bandwidth in high performance systems [13]. Finally, the lmbench is a suite of applications for measuring bandwidth and latency of various UNIX functionalities such as context switching, file system, memory, and network management [14].

Fig 6 and 7 shows the experimental results of the nbench and STREAM benchmarks, respectively. In each figure, x-axis represents the number of processes executing the benchmark independently while y-axis is the average runtime of processes. Our CPU intensity and memory intensity measurements suggest that the nbench can be categorized as a CPU-bound application while the STREAM as a memory-bound application. Hence, the nbench shows a relative better scalability than the STREAM in multicore NUMA systems, also as observed in Fig 3. In addition, we can notice that, for the STREAM benchmark, the proposed policy provides better performance than the conventional Linux kernel policy, by reflecting not only CPU load but also memory load for CPU allocation decisions. For the nbench benchmark, both policies show comparable results since it is a CPU-bound application.

Fig 8 depicts the performance evaluation results when we run the nbench and STREAM benchmarks concurrently with the various number of processes. The goal of this experiment is analyzing the effect of the proposed policy when

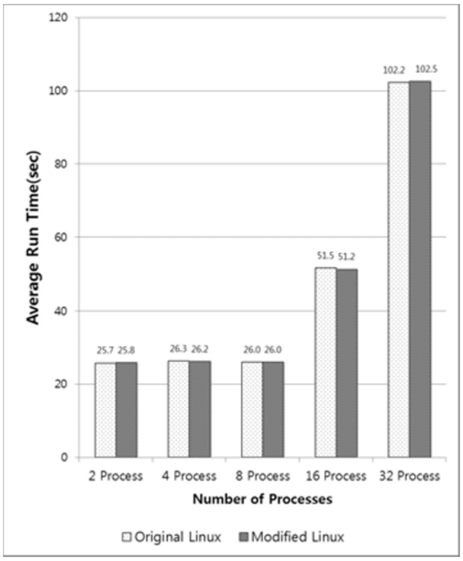

Fig. 6. nbench benchmark results

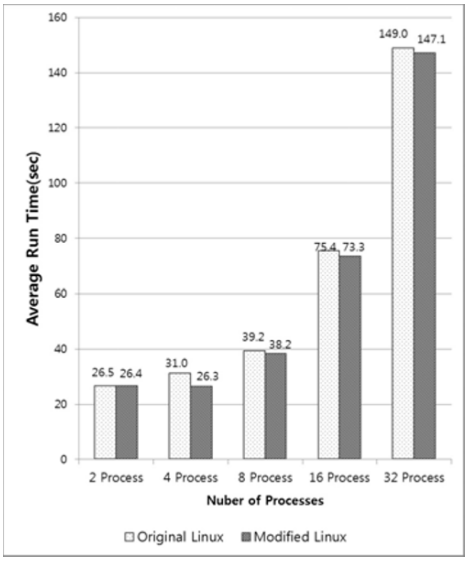

Fig. 7. STREAM benchmarks results

there exist heterogeneous applications together. The results also show that our proposed policy performs better than the conventional one. We expect that when we apply not only the CPU load and memory load but also the characteristics of a process such as CPU-bound and memory-bound into CPU allocation decisions, we can obtain more performance improvements.

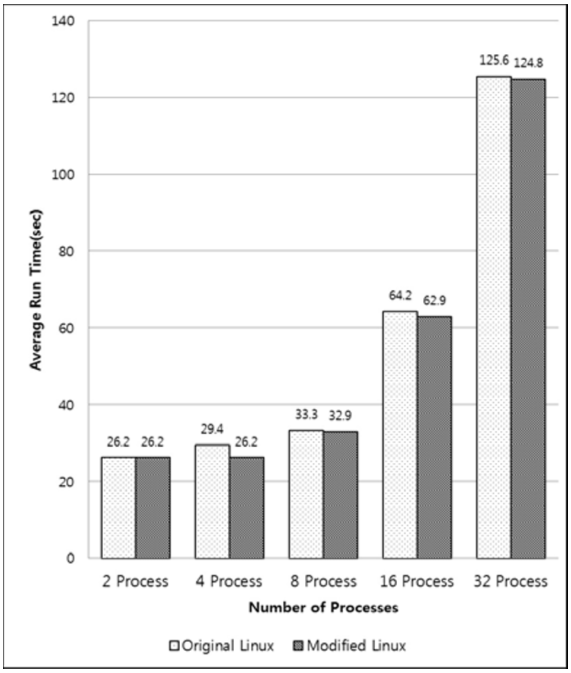

Fig. 8. nbench and STREAM benchmarks results

Fig 9 shows the lmbench results. The lmbench benchmark consists of a set of applications testing a specific part of UNIX operating systems such as process management latency, networking latency, memory read/write bandwidth, context switching latency, STREAM latency, and memory read latency. In this experiment, we create processes ranging from 2 to 64 where each process executes an application independently. The results display that some applications such as memory read/write bandwidth and STREAM latency gain performance benefits from our policy, while other applications does not show marginable performance differences. Detailed analysis reveals that the latter applications are mainly CPU-bound, which leads to little performance differences.

Our final experiment is the possibility of applying process characteristics on CPU allocations. In this experiment, we have run several processes, one half executes the nbench application (a CPU-bound application) and the other half

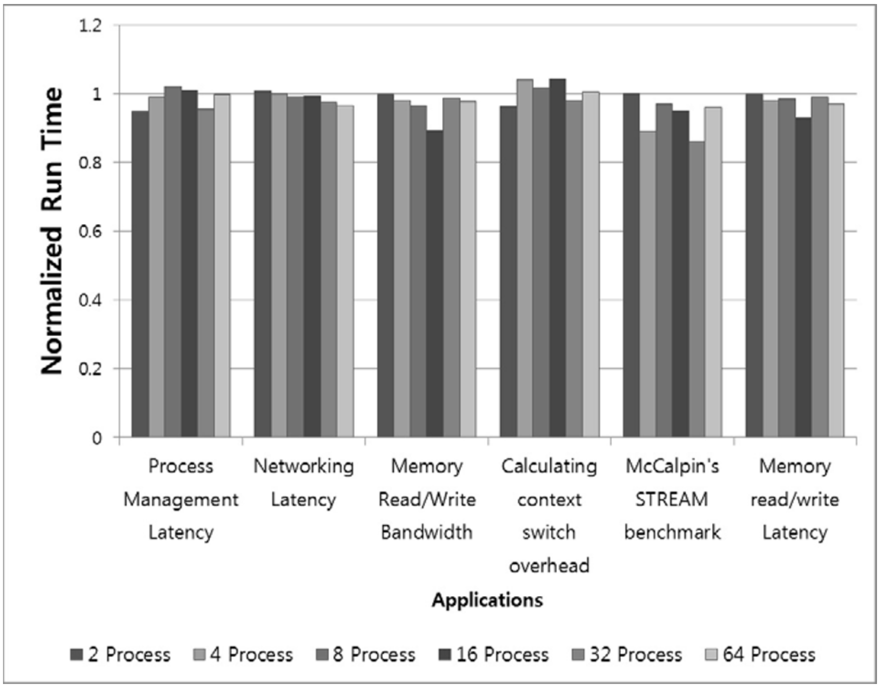

Fig. 9. lmbench benchmark results

executes the STREAM application (a memory-bound application). Then, we devise three CPU allocations, namely Node-bias, CPU-bias, and CPU-unbias, as described in Fig 10. The Node-bias allocation statically assigns CPU-bound processes into one node and memory-bound processes into the other node, separately. In the CPU-bias allocation, each core with even number has CPU-bound processes only while other core with odd number has memory-bound processes. Finally, in the CPU-unbias allocation, each core has both CPU-bound and memory-bound processes evenly. The results have shown that the Node-bias allocation performs the worst since assigning memory-bound processes into the same node causes a lot of remote memory accesses and the bus contentions. Interestingly, the CPU-bias allocation shows better performance than the CPU-unbias allocation, implying that, in the same node, assigning same characteristic processes into the same CPU might reduce the bus contentions among CPUs. Currently, we are investigating this issue more closely with a logic analyzer that can measure the bus contentions quantitatively.

5 CONCLUSION

In this paper, we have proposed a new memory-aware CPU allocation policy. Using CPU intensity and memory intensity of a process, it estimates

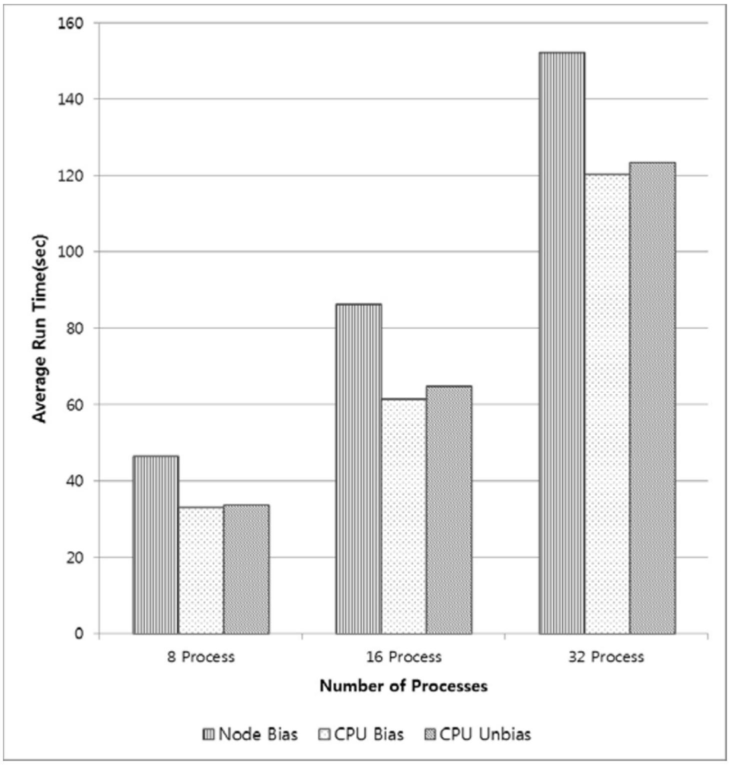

Fig. 10. Applying process characteristics on CPU allocations

CPU load and memory load of a core and applies them for CPU allocation decisions hierarchically and adaptively. Implementation based experimental results have shown that the proposed policy indeed enhances the response times of benchmarks.

As the popularity of multicore NUMA systems increases in computer architecture areas, operating system researchers also study internal structures and policies related to the systems including scheduling [15] [16] [17] [18] [19], synchronization [20] and energy efficiency [21]. Our proposal is one of those studies, addressing CPU allocation issues to reduce the number of remote memory references.

We are considering two research directions as future works. One direction is analyzing the optimal value of the control parameter α under the various conditions such as number of processes, characteristics of each process, and CPU and memory capabilities. Another direction is conducting more thorough quantitative experiments with diverse benchmarks.

Acknowledgement

This research was supported by National IT Industry Promotion Agency(NIPA) under the program of Software Engineering Technologies Development and Experts Education.

References

[1] Intel Multi-core Technology, http://www.intel.com/multi-core/.
[2] AMD Magpy-Cours, http://developer.amd.com/zones/ magny-cours/Pages/default.aspx.
[3] ARM Cortex-A9 processor, http://www.arm.com/products/ processors/cortex-a/cortex-a9.php.
[4] Sunpyo Hong and Hyesoon Kim: An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness. in Proceeding of the 36th Annual International Symposium on Computer Architecture (ISCA), pp. 152-163, (2009)
[5] Burton Smith: Many-core Operating System Challenges and Opportunities. in the 3rd Workshop on the Interaction between Operating Systems and Computer Architecture, (2007)
[6] Joab Jackson: Multicore chips require OS rework, Windows architect advises. InfoWorld, http://www.infoworld.com/d/developer-world/multicore-chips-requires-os-rework-windows-architect-advises-182. (2010)
[7] Andrew S. Tanenbaum: Modern Operating Systems. 3rd edition, Pearson Prentice Hall (2009)
[8] Wolfgang Mauerer: Professional Linux Kernel Architecture. WILEY. (2008)
[9] M.Tim Jones: Inside the Linux 2.6 Completely Fair Scheduler. IBM Research Center, http://www.ibm.com/developerworks/linux/library/l-completely-fair-scheduler (2009)
[10] Martin J. Bligh, Matt Dobson, Darren Hart, Gerrit Huizenga: Linux on NUMA Systems. in the Proceedings of the Linux Symposium (2004)
[11] Changes in the Linux Scheduler as of 2.6.23, http://gustavus.edu/+max/os-book/updates/CFS.html.
[12] nbench, http://www.tux.org/ mayer/linux/bmark.html
[13] STREAM benchmark, http://www.cs.virginia.edu/stream.
[14] Carl Staelin: lmbench - an extensible micro-benchmark suite. HP Laboratories Israel (2004)
[15] Tong Li, Dan Baumberger, David A. Koufaty and Scott Halm: Efficient Operating System Scheduling for Performance-Asymmetric Multicore Architecture. in the SC conference (2007)
[16] M. Correa, A. Zotzo and R. Scheer: Operating System Multilevel Load Balancing. in the ACM SAC conference, pp 1467 1471 (2006)
[17] Rafeal Chanin, Monica Correa, Paulo Fernandes, Afonso Sales, Roque Scheer, Avelino F. Zorzo: Analytical Modeling for Operating System Schedulers on NUMA systems. Electronic Notes in Theoretical Computer Science, 131 149 (2006)

[18] Joseph Antony, Pete P. Janes, and Alistair P.Rendell: Exploring Thread and Memory Placement on NUMA Architectures: Solaris and Linux, UltraSPARC/FirePlane and Opteron/HyperTransport. High Performance Computing (2006)
[19] Mathieu Faverge, Pierre Ramet: A NUMA Aware Scheduler for a Parallel Sparse Direct Solver. Workshop on Massively Multiprocessor and Multicore Computers (2009)
[20] Christoph Lameter: Effective Synchronization on Linux/NUMA Systems. Gelato Conference (2005)
[21] Andreas Merkel and Frank Bellosa: Memory-aware Scheduling for Energy Efficiency on Multicore Processors. in the workshop of 2008 USENIX Hotpower (2008)