Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Asymmetry-aware load balancing for parallel applications in single-ISA multi-core systems*

Eunsung KIM, Hyeonsang EOM‡, Heon Y. YEOM

(*School of Computer Science and Engineering, Seoul National University, Seoul 151-744, Korea*)

E-mail: eskim@dcslab.snu.ac.kr; hseom@cse.snu.ac.kr; yeom@snu.ac.kr

Received July 4, 2011; Revision accepted Mar. 27, 2012; Crosschecked May 4, 2012

Abstract: Contemporary operating systems for single-ISA (instruction set architecture) multi-core systems attempt to distribute tasks equally among all the CPUs. This approach works relatively well when there is no difference in CPU capability. However, there are cases in which CPU capability differs from one another. For instance, static capability asymmetry results from the advent of new asymmetric hardware, and dynamic capability asymmetry comes from the operating system (OS) outside noise caused from networking or I/O handling. These asymmetries can make it hard for the OS scheduler to evenly distribute the tasks, resulting in less efficient load balancing. In this paper, we propose a user-level load balancer for parallel applications, called the 'capability balancer', which recognizes the difference of CPU capability and makes subtasks share the entire CPU capability fairly. The balancer can coexist with the existing kernel-level load balancer without detrimenting the behavior of the kernel balancer. The capability balancer can fairly distribute CPU capability to tasks with very little overhead. For real workloads like the NAS Parallel Benchmark (NPB), we have accomplished speedups of up to 9.8% and 8.5% in dynamic and static asymmetries, respectively. We have also experienced speedups of 13.3% for dynamic asymmetry and 24.1% for static asymmetry in a competitive environment. The impacts of our task selection policies, FIFO (first in, first out) and cache, were compared. The use of the cache policy led to a speedup of 5.3% in overall execution time and a decrease of 4.7% in the overall cache miss count, compared with the FIFO policy, which is used by default.

Key words: Scheduler, Load balancing, Capability asymmetry, OS noise, Multi-core
doi:10.1631/jzus.C1100198 **Document code:** A **CLC number:** TP316

1 Introduction

Fast advancement of processor technology increases the degree of on-chip parallelism, which allows processors to contain tens and even hundreds of cores. Multi-core processors are widely used in embedded, desktop, or server computing to accelerate computation in a parallel manner and to decrease energy consumption efficiently. Among the various multi-core architectures our main focus is a single-ISA (instruction set architecture) multi-core system. This system can be subdivided into the following types: symmetric multi-core (SMC) processors like Intel Core 2, AMD Phenom, or Parallax Propeller processors having multiple cores that are all exactly the same. Every single core has the same architecture and the same capability. In asymmetric multi-core (AMC) processors like Intel Sandy Bridge or AMD Fusion, the chips have multiple cores onboard, which may have different designs and capabilities.

Commodity operating systems do not have any bias toward one CPU with respect to the others when distributing tasks among all the CPUs (Bovet and Cesati, 2005) and eventually making each CPU execute the same number of tasks. This enables operating systems (OSs) to fully exploit the parallelism of the

‡ Corresponding author

* Project supported by the Next-Generation Information Computing Development Program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (No. 2011-0020521) and the Korea Communications Commission, under the Communications Policy Research Center Support Program supervised by the Korea Communications Agency (No. KCA-2011-1194100004-110010100)

processors and improves the performance of the system. But the approach is based on the premise that a system consists of two or more identical CPUs. Consequently, the OSs can cause some problems in the presence of asymmetry in CPU capability. The growth of device speed and the advent of asymmetric processors increase the level of capability asymmetry.

We can categorize asymmetry into three classes: natural, dynamic, and static. Natural asymmetry of CPU capability results from the default action for OS-inside noise such as timer interrupt, translation lookaside buffer (TLB) invalidation, or page fault. OS-inside noise is essential for maintaining a system; in various domains, especially the supercomputing domain, there has been increasing attention to this noise (Petrini *et al*., 2003; Gioiosa *et al*., 2004; Beckman *et al*., 2006; 2008; De *et al*., 2007). OS-inside noise is not considered in this paper, however, as it has a negligible impact on application performance in most single-ISA multi-core systems. Natural asymmetry is the baseline for our performance evaluation. Dynamic asymmetry of CPU capability comes from OS-outside noise like network processing or disk I/O. Outside noise may have great effect on application performance, depending upon its duration and frequency. Our focus is on network processing because its execution requires more CPU time than disk I/O, as will be described in Section 2. Static asymmetry of CPU capability is caused by the different capabilities of cores in an AMC processor. AMC was proposed as a more power efficient alternative to SMC (Kumar *et al*., 2004) and can potentially deliver a higher performance per watt than SMC (Kumar *et al*., 2003; Hill and Marty, 2008).

CPU scheduling subsystems of modern OSs have independent per-CPU runqueues and provide two main functionalities: task scheduling and load balancing. The functionalities attempt to provide equal CPU time to each task inside a runqueue or among runqueues. Here we inspect the impact that the asymmetry of CPU capability has on the load balancing functionality. Current load balancing mechanism utilizes the number of tasks in the runqueue as the load metric on each CPU and balances the system-wide load by periodically migrating tasks from longer runqueues to shorter ones. We call it the 'runqueue balancer'. The balancer maintains comparatively well the fairness on CPU time to each task in the absence of asymmetry in CPU capability; however, it fails to distribute CPU time evenly to tasks under the opposite situation. For example, it may make some tasks run more frequently than others on CPUs with heavy OS noise or on slow CPUs of AMC systems. This unfairness leads to low performance on parallel applications, a technique intensively used in the scientific domain now and also expected to be often utilized in various commercial domains in the near future (Asanovic *et al*., 2009).

In this paper, we explore the source of CPU capability asymmetry in a single-ISA multi-core system, define metrics to measure CPU capability, and propose the asymmetry-aware user-level load balancer, capability balancer, for parallel applications. The capability balancer recognizes the difference of CPU capability and makes subtasks share the entire CPU capability fairly. For capability balancing, we define the capability of a CPU as the number of instructions available during a period, from which two load metrics, effective capability and effective capability per task, are derived. The metrics enable an efficient detection of capability asymmetry and fair use of CPU capability. In the multi-tasking environment, we currently employ two task selection policies, FIFO (first in, first out) and cache, to choose tasks to be migrated. The FIFO policy is based on task order in the runqueue, and the cache policy considers the cache footprint of tasks. The capability balancer is completely non-intrusive to the existing kernel load balancer and can coexist with it. The user-level balancer is suitable for parallel applications while the kernel-level balancer is good for current commercial and end-user multi-programmed workloads. Users can easily and freely choose between these two balancers.

To achieve a trade-off between responsiveness and overhead of the capability balancer, we evaluate the impact of using various intervals to determine the optimal balance interval. We select one second as the balance interval in the experiment. By experimenting to compare the performance and fairness of the runqueue balancer and the capability balancer using a synthetic program, we conclude that the capability balancer can fairly distribute CPU capability to tasks, with very little overhead. Furthermore, we evaluate our balancer for the NAS Parallel Benchmark (NPB) as the real workloads. We experience a slowdown of about 3% for natural asymmetry but speedups of up to 9.8% and 8.5% with the use of dynamic and static asymmetries, respectively. We also experience

speedups of 13.3% for dynamic asymmetry and 24.1% for static asymmetry in a competitive environment. Finally, we compare the impact of the task selection policies. The use of the cache policy leads to a speedup of 5.3% in overall execution time and a decrease of 4.7% in the overall cache miss count, compared to the FIFO policy.

2 Problem statement

Modern multiprocessor operating systems, such as Linux, Solaris, FreeBSD, and Windows, support multiprocessor machines with many different flavors of classic multiprocessor architecture, like multi-threaded or multi-core. In general, the OSs are composed of a CPU scheduling subsystem, a memory management subsystem, a filesystem, and a networking subsystem.

The CPU scheduling subsystem aims to share CPU resource efficiently. The subsystem is composed of multiple schedulers which administer the corresponding CPU's runqueues, rather than a single scheduler to supervise all CPUs (Fig. 1). The reason is that, the scalability problem arises as the number of CPUs increases. The schedulers operate independently and cooperate with others via shared variables. Each scheduler manages the lifecycle of tasks inside its runqueue using two major functionalities: task scheduling and load balancing. Task scheduling distributes CPU time to tasks in the runqueue according to a scheduling policy, for instance, FIFO, round-robin, or multilevel queue. Load balancing migrates tasks among the runqueues using some load criteria. To do this, it decides when to balance the load, which CPUs need load to be relieved, or how many tasks to migrate under different conditions. Here, we focus on the process of load balancing.

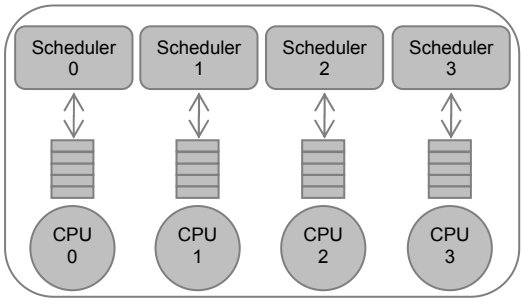

Fig. 1 CPU scheduling subsystem in a multiprocessor OS

The design of load balancing mechanisms in contemporary OSs incorporates the following assumptions: (1) The capability of CPU on the system is the same; (2) Tasks are independent of each other. Based on these assumptions, OSs try to achieve two compatible objectives to stabilize the load of the system. One is to prevent CPUs from being idle while other CPUs still have tasks waiting to be executed. The other is to keep the difference among the numbers of ready tasks on all CPUs as small as possible. Consequently, most OS schedulers concentrate their efforts on balancing the numbers of tasks in runqueues (the runqueue lengths) among multiple CPUs. This is why it is called the 'runqueue balancer'.

The runqueue balancer operates according to the number of tasks in the system (Fig. 2). It preserves the soft CPU affinity of tasks if the number of tasks is smaller than the number of CPUs in the system. Soft affinity is the tendency of a scheduler to try to keep tasks on the same CPU as long as possible to avoid corruption of the cache. If the number of tasks is a multiple of the number of CPUs, it does not balance the load because the system has already been balanced. If the above conditions are not met, however, it attempts to balance the load.

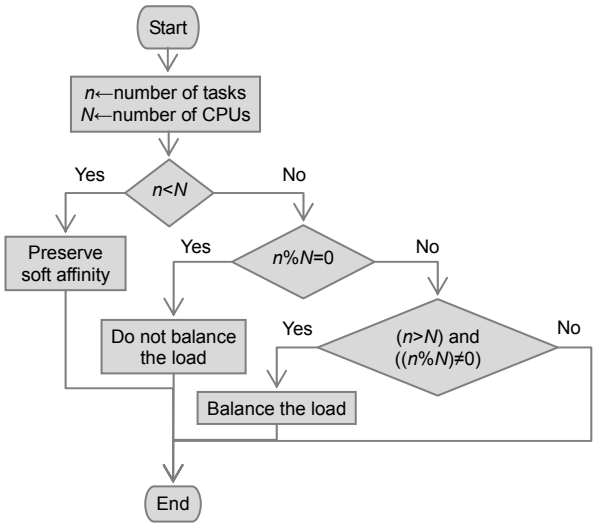

Fig. 2 Operation of the runqueue balancer

Parallel applications such as MPI, OpenMP, and Unified Parallel C (UPC) have dependency among their subtasks. They are often used in high performance computing such as scientific research or computational science, and expected to be used in various commercial domains in the near future (Asanovic *et*

al., 2009; Gioiosa *et al*., 2010). The performance of parallel applications is dominated by the speed of the slowest subtask because each subtask may have dependency on one another. That is, a parallel application runs as fast as its slowest subtask. Therefore, users often run parallel applications in dedicated environments where their subtasks can receive equal CPU times, and hence the runqueue balancer does not balance the subtasks as they have already been balanced. This works well in the system with symmetric CPUs but is likely to fail with fast devices or on the recent AMC systems.

OS noise is an incidental but essential activity of OS, irrespective of the tasks being executed in its system. It occurs inside OS in the form of periodic timer interrupts, page faults, or cache invalidation (OS-inside noise). It also comes from external devices like network interface cards or hard disks (OS-outside noise). OSs handle the noise with higher priority than normal tasks, which in turn delays the execution. The CPU scheduling subsystem does not consider the delay of normal tasks caused by OS noise and thus is agnostic to OS noise.

OS-inside noise is indispensable to operate a system and the supercomputing domain raises attention to noise (Petrini *et al*., 2003; Gioiosa *et al*., 2004; Beckman *et al*., 2006; 2008). However, it generally terminates very fast with little effect and can be neglected in prevalent single-ISA multi-core systems. We name it 'natural capability asymmetry' and use it as the basis of our performance evaluation.

OS-outside noise by network or disk processing may persist its execution for a very long time in case of heavy traffic or I/O. Network traffic is processed entirely via a CPU except for network cards with a special hardware like TOE (TCP offload engine), whereas I/O data is processed by a disk controller. So, network processing consumes more CPU time than disk I/O processing. It is also more influential on application performance. We focus on network processing in this respect. The gigabit network environment is one of major sources of heavy network traffic. Basically, only a specific CPU is used to handle the network traffic although there are multiple CPUs in the system. This is to avoid cache corruption of each CPU as well as the packet reordering problem (Salim, 2001). Therefore, a task running on the CPU that handles the network traffic experiences asymmetry in its CPU capability when there is heavy network traffic. This results in poor application performance. We call the asymmetry by OS-outside noise 'dynamic capability asymmetry'.

AMC processors (Kumar *et al*., 2004) were proposed as a more power efficient alternative to SMC processors, which use identical cores. An AMC processor would contain cores that expose the same ISA, but differ in features, size, power consumption, and performance. A typical AMC processor would contain cores of two types: fast and slow (Kumar *et al*., 2003). Fast cores are characterized by complex super-scalar out-of-order pipelines, aggressive branch-prediction, pre-fetching hardware, and high clock frequency, whereas slow cores have a simple in-order pipeline, less complex hardware, and a lower clock speed. Fast cores occupy a larger area and consume more power than slow cores. Thus, a typical system would contain a small number of fast cores and a large number of slow cores. The asymmetry that comes from the gap among CPU capabilities is named 'static capability asymmetry'.

The current runqueue balancer does not consider the change of CPU capability. So, it does not migrate the subtasks of parallel applications even if it has difference in CPU capability. The performance of parallel applications under the control of the runqueue balancer is bounded to the speed of the subtask on the CPU with the minimum capability, and is defined as

$$\text{Capability\_Set(CS)} = \{C_1, C_2, ..., C_n\}, \quad (1)$$

where C_i is the capability of CPU i and n is the number of CPUs in a system.

The performance of parallel applications under the runqueue balancer is

$$\min(\text{CS}). \quad (2)$$

In this paper, we seek to evenly balance the total CPU capability to subtasks and therefore the performance of parallel applications is the average of the overall CPU capability. We can theoretically achieve the same performance as the runqueue balancer in case of natural asymmetry and better performance than under static and dynamic CPU capability asymmetries, such as

$$\text{avg(CS)} \geq \min(\text{CS}). \quad (3)$$

In this context, we argue that multi-core systems must deal with the asymmetry of CPU capability since the increase of device speed enlarges the impact of OS-outside noise and the advent of the new processors introduces non-identical CPU capability.

3 Asymmetry-aware load balancing

We introduce user-level capability balancing for parallel applications which understand CPU capability asymmetry, and describe the design and implementation of the balancer. In the rest of this paper, our discussion will be focused on Linux (kernel version 2.6.32.5). Note that our approach can easily be applied to other OSs.

3.1 Criteria of capability balancing

We provide a user-level load balancer that makes subtasks of a parallel application fairly utilize the overall CPU capability. The balancer does not require any application modifications and is independent of application implementation details. It consists of per-CPU balancers, each of which runs on a CPU, operates independently, and communicates with one another via shared variables. The per-CPU balancer wakes up periodically, checks for asymmetry, pulls tasks from another CPU to its CPU if possible, and then sleeps again. The period over which the balancer sleeps, i.e., the balance interval, determines the frequency of migrations or its overhead. The choice of this parameter will be illustrated in Section 4.2.

Unlike the runqueue balancer which uses runqueue length as the load indicator, we use the capability of CPU as the index. To model the capability as shown in Fig. 3, we first divide CPU time available in a balance interval into three classes:

$$\text{CPU\_Time}=\text{User\_Time}+\text{Noise\_Time}+\text{Idle\_Time}. \quad (4)$$

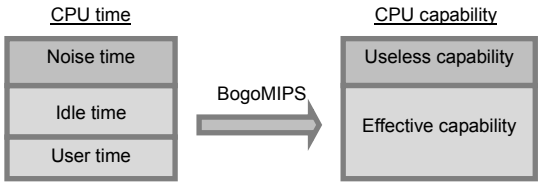

Fig. 3 CPU capability modeling

User_Time is CPU time consumed by tasks, Noise_Time is CPU time consumed by OS noise, and Idle_Time is CPU time when it is doing nothing. CPU_Time is calculated by the clock tick count in most OSs. In an AMC system with a clock source, slow CPUs report the same CPU time as fast CPUs in an interval, although they have done less work than fast ones. To accurately account for CPU usage, we use BogoMIPS (bogus millions of instructions per second), which gives some indication of the processor speed while being the only portable way over the various CPUs (Intel-type and non-Intel-type) for obtaining an indication of CPU speed. Consequently, we define the capability (C) of a CPU in a balance interval as follows:

$$C=\text{CPU\_Time}\times\text{BogoMIPS}. \quad (5)$$

C denotes the total number of instructions executed in a balance interval. Next, we define the effective capability (EC) of a CPU as follows:

$$\text{EC}=(\text{User\_Time}+\text{Idle\_Time})\times\text{BogoMIPS}. \quad (6)$$

EC represents the number of effective CPU instructions available to tasks rather than OS noise. It includes the instructions consumed during CPU idle time. Finally, we define the effective capability per task (ECPT) of a CPU as follows:

$$\text{ECPT}=\begin{cases}\dfrac{\text{EC}}{\text{task\_num}}, & \text{if task\_num}>0,\\ \text{EC}, & \text{otherwise.}\end{cases} \quad (7)$$

ECPT depicts the number of CPU instructions accessible to a task in a CPU. It enables the detection of capability asymmetry.

3.2 Balancing process

The shared vector variable, ecpt_vec, in capability balancing, maintains ECPT values from each per-CPU balancer. We introduce a single lock to the variable to synchronize the balancing process among them. The per-CPU balancer carries out the following process at each interval:

1. It computes ECPT for its local CPU over the elapsed balance interval.

2. It acquires the lock to the ecpt_vec and obtains ECPTs for the remote CPUs.

3. It concludes that there is capability asymmetry in the system if its ECPT is greater than the average ECPT over all CPUs. Otherwise, go to step 8.

4. It finds the remote CPU that has the smallest ECPT, which has not recently been involved in a migration. If it finds nothing, go to step 8.

5. It selects a task from each of the local and the remote cores according to a task selection policy.

6. It switches the two tasks.

7. It updates its ECPT to the vector.

8. It releases the lock.

There are two task selection policies in current implementation: FIFO (by default) and cache. The FIFO policy chooses a task in the order of the runqueue of a CPU. Balancers bounce the same task among multiple CPUs at each interval, thus avoiding the ping-ping effect. The cache policy picks a task with the smallest cache footprint in a CPU cache.

In load balancing, it is important to consider the impact of cache invalidation by task migration. Cache invalidation causes a failed attempt to read or write a piece of data in the cache, which results in much longer latency and poor performance in main memory access. To tackle the problem, the runqueue balancer of the current Linux scheduler, CFS, checks whether a migrated task is cache-hot or not. A task is hot if it is a buddy (a task following or preceding the currently running task in the order imposed by the CFS algorithm) or if the time elapsed since the start time of the execution does not exceed a threshold.

Cache footprint typically describes the portion of a cache used by a task. It incurs heavy overhead to obtain the value from the OS during runtime. So, we use the working set as the indicator of the cache footprint. The working set represents the amount of memory currently marked as referenced, or accessed by a task and is available at runtime. It thus enables a much better estimation of cache footprint, compared with the resident set which describes the portion of a process's memory that is held in memory. Two tasks with smaller working sets may be executed in separate time slices on a CPU without evicting each other's cache lines. On the other hand, there may be another task with a larger working set which performs better via cached data reuse when running on its own CPU. However, it would suffer from performance loss if another task starts on the CPU and hence starts evicting cache lines that the first program might eventually reuse. Consequently, cache footprint can be well reflected by the working set. Under the cache policy, we select a task with the smallest working set to minimize the effect of invalidating cache lines of other tasks.

3.3 Implementation

Capability balancing (Fig. 4) is composed of a command utility (chcb), a standalone multiprocess program (capability-balancer), and a loadable kernel module (cbinfo). chcb and the capability-balancer are implemented in Python for easy development and portability. When capability balancing is applied to other OSs, only cbinfo needs to be rewritten.

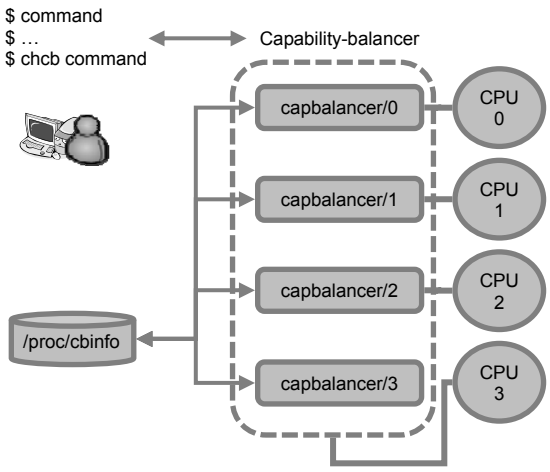

Fig. 4 Capability balancing modules

The chcb utility takes the statement to execute an application as input and forks the application to join in capability balancing. It then pins its tasks on each CPU in the order of ECPT of CPU. When the application finishes, chcb prints out its stdout or stderr on the screen and excludes the application from the control of capability balancing. A user has to start different chcb if he/she wants to run a workload that consists of different parallel applications.

The capability-balancer consists of capbalancer/*n* (where *n* is the logical number of its local CPUs) to run on each CPU in the system. The capbalancer/*n* is responsible for migrating the tasks according to the capability change. It is implemented as a process instead of a thread. The restriction is enforced by the Python GIL (Global Interpreter Lock),

which must be acquired for a thread to enter the interpreter's space. The GIL allows only one thread to be executed within the Python interpreter at one time. Python multiprocessing allows the side-stepping of the GIL. We also give a real-time priority to capbalancer/*n*, using the chrt command for fast response.

The capinfo provides the information required by the capability-balancer. It is implemented as a loadable kernel module (LKM) integrated into the '/proc' filesystem and creates a virtual file in the filesystem. It organizes various kernel data and enables acceleration of user-level balancing with little overhead. It also maintains the pid list of each runqueue per process group, the working set size of each pid, and so on.

At the user level, it is difficult or impossible to migrate tasks by directly manipulating the information of kernel runqueues. Instead, we use the hard CPU affinity which makes tasks adhere to specified CPUs. The affinity is realized by the sched_setaffinity system call in Linux. The system call changes the cpu_allowed vector of a task. On the system call, if a task is not on a runqueue and is not running, then it will be properly placed at the next wake-up. Otherwise, it needs help from the migration thread. The migration thread is a high priority kernel thread that performs task migration by bumping a task off a CPU and then pushing it onto another CPU. A task migrated using the system call is fixed to the new CPU. The runqueue balancer cannot move it during load balancing. Hence, we ensure that any tasks moved by capbalancer/*n* do not get moved again by the runqueue balancer. This also allows us to apply capability balancing to a particular application without preventing the runqueue balancer from balancing the load of any other unrelated tasks.

4 Experimental evaluation

This section describes the experiments we have performed to evaluate the effectiveness and validity of capability balancing under natural, dynamic, and static asymmetries.

4.1 Experimental setup

Our evaluation was conducted on the multiprocessor system with four symmetric cores. The specifications are shown in Table 1. This setup was used by default in the experiment for natural capability asymmetry (Fig. 5a). Additionally, we have unloaded unrelated daemon or services that may affect the balancers and have minimized other loads except for the parallel applications.

Table 1 System specification

Unit	Specification
Processor	AMD Phenom 9550 X4 Quad-Core (2.2 GHz per core)
L1 cache	64 KB (data)+64 KB (instruction) per core
L2 cache	512 KB per core
L3 cache	2 MB shared
Main memory	2 GB
NIC	Intel PRO/1000 PT server adapter (1 Gb/s)
OS	Debian GNU/Linux 6.0 (kernel 2.6.32.5)

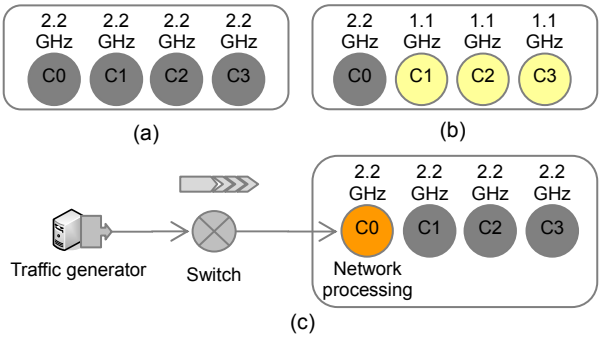

Fig. 5 Experimental configurations for natural (a), static (b), and dynamic (c) capability asymmetries

To evaluate static capability asymmetry, we built an AMC system from the SMC system. We utilized the Linux kernel CPUfreq subsystem (Brodowski, 2005; Pallipadi and Starikovskiy, 2006) to achieve this. It statically set the processor frequency or dynamically scaled the frequency based upon system load. The hardware limits of the current SMC were from 1.1 to 2.2 GHz and its available frequency steps included 2.2 GHz and 1.1 GHz. We configured the AMC system with a fast core (2.2 GHz) and three slow cores (1.1 GHz), as shown in Fig. 5b. In the experiments for dynamic capability asymmetry, we used network traffic as the OS-outside noise. We employed pktgen (Olsson, 2005) as a traffic generator. It generates network frames at very high speed because it operates in the kernel layer.

Recent network cards have the interrupt coalescing (IC) feature with which network frames are

collected and one single interrupt is generated for multiple frames to avoid flooding the host system with too many interrupts. This feature makes it especially hard to measure the effect of network traffic by the pktgen. To eliminate this, we set InterruptThrottleRate of the NIC to 0 (disabling the feature), which limits the number of interrupts per second. Fig. 6 shows the change of the CPU utilization of core 0 according to each packet rate. The growth of CPU utilization is in proportion to the increase of the packet rate when the packet rate is not larger than 10^8 packets/s. This is due to the fact that the traffic generator machine reaches the limit of its hardware capability for the packet rates over 10^8 packets/s. We used 10^7 packets/s as the worst case.

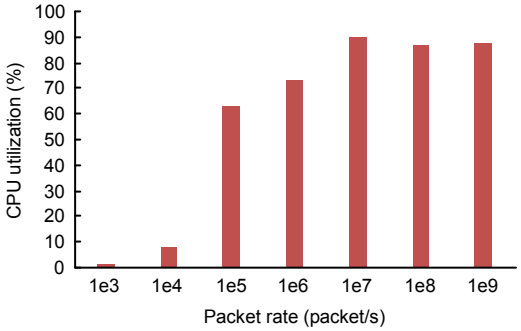

Fig. 6 CPU utilization according to the packet rate

The network traffic was processed by a specified CPU, as described in Section 2. The default configuration of our kernel was core 3, but for convenience we modified the smp_affinity file for the irq of the NIC in the /proc filesystem and made core 0 take charge of network processing. Fig. 5c depicts the configuration of the experiments.

Unlike the runqueue balancer, our user-level load balancer has to compete for CPU time with other normal tasks to make its balancing decision, and it may not guarantee to make a timely and precise decision. To minimize the effect of the problem, we configure the capability-balancer to be a real-time process using the chrt command. Real-time processes are never blocked by normal ones and have a short response time.

4.2 Microbenchmark

We introduce a synthetic MPI program, chore-counter, to compare capability balancing with runqueue balancing. The program is based on the OS noise measurement benchmark of Netgauge (Hoefler *et al.*, 2007). Each subtask of the program performs a very small work quantum (a chore) for 100 s and records how many times the chore has been carried out. In natural asymmetry the number of chores should be similar or equal for each subtask. In case of dynamic or static asymmetry, this number varies. After 100 s, the task computes the total number of chores, defined as task_chores. Finally the root task of the program calculates the overall average and the standard deviation of the task_chores, named avg_chore and stdev_chore, respectively. The avg_chore describes how well a load balancer distributes CPU capability to each task under various capability asymmetries. A higher avg_chore means that a load balancer balances tasks across CPUs more efficiently than other balancers. The stdev_chore is used to represent how fairly a load balancer distributes CPU capability to each task under various capability asymmetries. A lower stdev_chore means that a load balancer balances tasks across CPUs more fairly than other balancers. We utilize these two metrics in the comparison of load balancers.

The balance interval in load balancing is a very important factor because it determines the trade-off between responsiveness and overhead of the balancing. We have evaluated the impact of the intervals on capability balancing to seek the optimal balance interval. Fig. 7 shows the avg_chore and stdev_chore of the capability balancer for each balance interval under natural, dynamic, and static capability asymmetries. It tells that too short an interval results in low avg_chore due to the balancing overhead, whereas the opposite presents high stdev_chore by low balancing responsiveness. Thus, we select 1 s as the best balance interval and use it as the interval for all subsequent experiments.

As the load metric for load balancing, the runqueue balancer uses CPU runqueue length, but the capability balancer uses CPU capability. The capability metric directly captures the share of CPU capability to be received and used by a task, and can be easily adapted to capture behavior in an asymmetric system. It is simpler than using the queue length metric because the runqueue balancer requires weighting threads by priorities, which can have different effects on running time depending on the task mix and the associated scheduling classes.

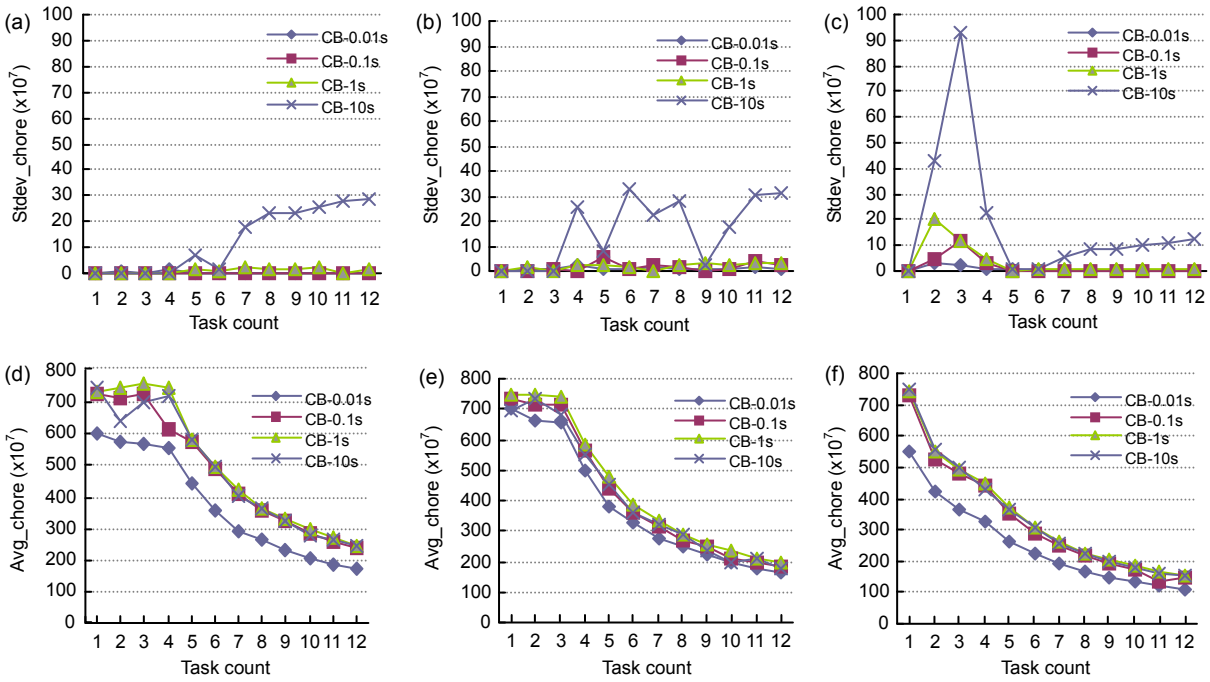

Fig. 7 Impact of each balance interval under natural (a, d), dynamic (b, e), and static (c, f) asymmetries

CPU capability based ECPT is a more elegant measure than the runqueue length, as it captures different task priorities and transient task behavior without requiring an additional factor like priorities.

We assume that the runqueue balancer faces the worst case for CPU capability compared with the capability balancer. So, the runqueue balancer runs at least one task on the network processing core for dynamic asymmetry, or does not run any task on the fastest core(s) for static asymmetry, when the number of its tasks is smaller than the number of cores. Fig. 8 shows the avg_chore and stdev_chore values of the capability balancer against the runqueue balancer. In the cases of all asymmetries, the capability balancer shows similar avg_chore to and lower stdev_chore than the runqueue balancer. This result indicates that our balancer can fairly provide CPU capability to tasks with very little overhead. Fig. 9 presents the difference of the number of task migrations of the runqueue balancer against the capability balancer made during the execution of synthetic workloads. The runqueue balancer tries to balance tasks only when the number of tasks is larger than the number of CPUs in the system and is not a multiple of the number of CPUs. It has a high migration count for these cases. The capability balancer starts migrating the tasks when it cannot fairly allocate CPU time to them. It then continues to balance CPU capability. The capability balancer has a large impact on cache invalidation in a competitive environment where multiple parallel applications run simultaneously. We solve the problem using the cache policy presented in Section 3.2.

4.3 Realistic workloads

Typically, the best performance of parallel applications can be achieved when each subtask of the application is dedicated to a CPU and repeatedly engages in a computational phase. After that it synchronizes or communicates with its peers. Every phase is followed by a barrier to ensure safe access to shared data structure. It can also be followed by a communication period to exchange data that is needed for the next computation. This means that no subtask is allowed to continue to the next computational phase until all its peers have completed the previous one. Thus, if one subtask is late, all its peers must stand idle until the former catches up. This process dramatically amplifies the impact of any local disturbances that prevent the subtask from completing its work on time. In this setting, the synchronization or communication can promise the best performance

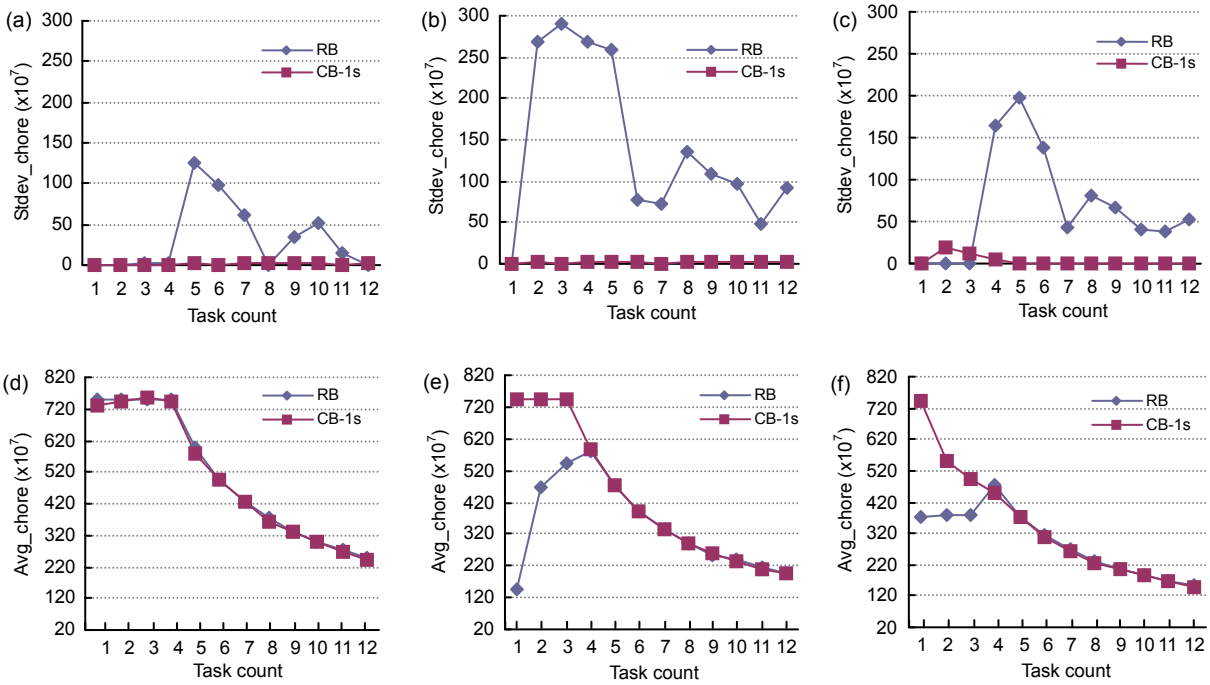

Fig. 8 Comparison of the capability balancer and the runqueue balancer under natural (a, d), dynamic (b, e), and static (c, f) asymmetries

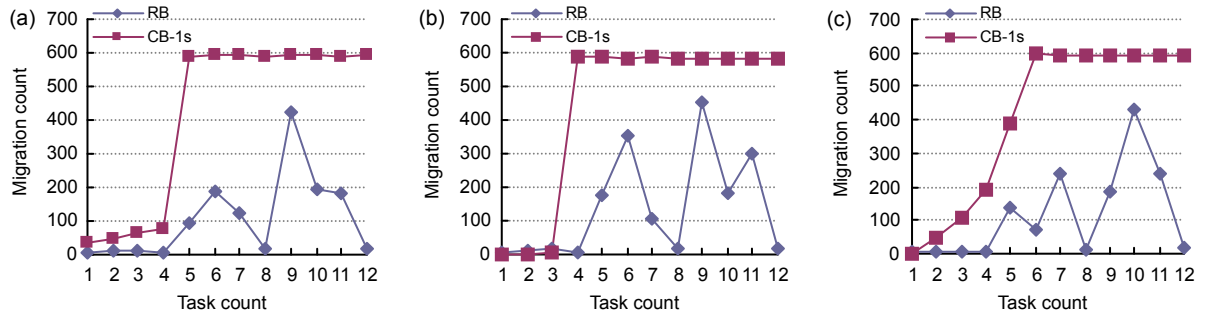

Fig. 9 Comparison of the migration count under natural (a), dynamic (b), and static (c) asymmetries

of parallel applications roughly at the same time. So, the applications run mainly in a dedicated environment for their tasks to be able to execute simultaneously on different processors or utilize such a specific technique as gang scheduling or coscheduling for better performance. Therefore, users can generally expect symmetry; that is, each subtask will gain the same capability from its CPU. However, the fantasy is often shattered by OS noises like network traffic and asymmetric hardware, which lead to longer synchronization. The capability balancer provides an elegant solution to the problem.

To validate the value of capability balancing for realistic workloads, we performed experiments of the NAS Parallel Benchmarks (Bailey *et al*., 1991). It includes the OpenMP implementation (NPB-OMP) for natural and static capability asymmetries, and the MPI implementation (NPB-MPI) for dynamic capability asymmetry. We conducted the NPB-OMP experiment on the configurations of Figs. 5a and 5b, respectively. We used four threads per benchmark and made each thread run on its own core.

Fig. 10 shows the speedup of the capability balancer against the runqueue balancer under natural

and static capability asymmetries. We have experienced a slowdown of about 3% for natural asymmetry and the overhead results from the management activity of capability balancing. However, we have accomplished a maximum speedup of 8.5% in static asymmetry, except for is.C and mg.B, which have short runtime and benefit little from capability balancing.

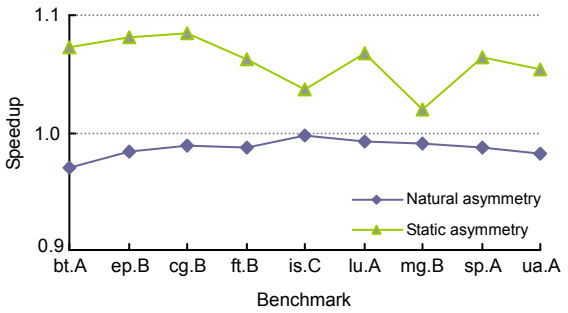

Fig. 10 Speedup of the capability balancer against the runqueue balancer

For the NPB-MPI experiment, we used the experimental node and another node to make use of MPI communication as OS-outside noise. We configured eight processes per benchmark and each node executed four processes. We excluded three benchmarks because bt.A and sp.A require the number of their subtasks be a square (1, 4, 9, …) and there is no MPI version of ua.A. Unlike the microbenchmark, we did not modify the default configuration of the experimental node: network traffic is handled in core 3 and the IC feature of the network card is enabled. Table 2 shows the total and average amounts of network traffic transferred during the execution of each benchmark.

Table 2 Amount of network traffic in NPB

Benchmark	Total volume (Mb/s)	Average volume (Mb/s)
ep.B	0.032	0.004
cg.B	2341	352
ft.B	5814	746
is.C	2884	788
lu.A	744	224
mg.B	186	67

Fig. 11 shows the speedup under dynamic asymmetry. We achieved a maximum speedup of 9.8%. Capability balancing is more efficient than runqueue balancing as long as the increase of the average traffic is concerned. However, the is.C gains less benefit due to the short execution time, although it generates much network traffic. The ep.B which runs in parallel has little data exchanged by its subtasks and cannot benefit from capability balancing.

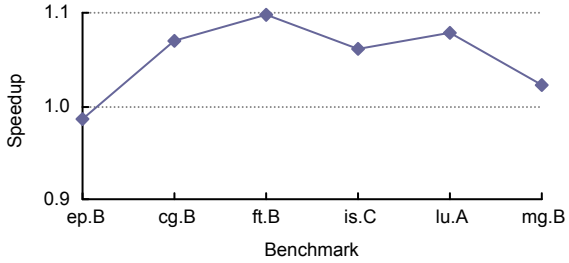

Fig. 11 Speedup of the capability balancer under dynamic asymmetry

4.4 Competitive environment and impact of the task selection policy

To demonstrate the strength of capability balancing in a competitive environment, we ran concurrently all the benchmarks used in Section 4.3, and compared the elapsed time until they were all completed under each load balancer. The results for ft.B and is.C were excluded in our experiment because they were executed with too large data structures, causing thrashing on the machine.

Thrashing results in constant page faults, which drastically slow down the system and disturb the operation of the capability balancer. Fig. 12 describes the result of the experiment, in which we have achieved a speedup of about 9.9% even in natural asymmetry. The runqueue balancer does not migrate tasks if the number of tasks is a multiple of the number of CPUs, as described in Section 2. Load balancing based on runqueue lengths cannot fully utilize CPUs available in the system, whereas the capability balancer can. We have experienced speedups of about 13.3% and 24.1% under dynamic and static asymmetries, respectively.

Task migration balances the system load but may incur another overhead. We consider cache invalidation as the main origin of the overhead. Loss of cache context for memory intensive benchmarks increases the cost of migrations to several milliseconds. Therefore, we conclude that the reduction of the

overhead incurred by task migration may lead to better performance. To achieve this, we provide the cache policy which estimates the cache footprint via the working set size of a task. Table 3 gives the statistics for working set size of NPB benchmarks, obtained by sampling the /proc filesystem every second during the lifetime of benchmarks.

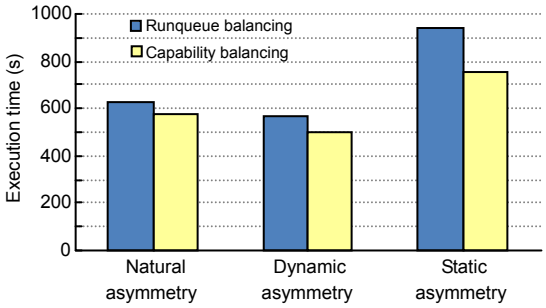

Fig. 12 Total execution time in a competitive environment

Table 3 NPB working set size

Working set	Size (MB)			
	Total	Average	Minimum	Maximum
bt.A	1606	45	45	45
ep.B	166	5	5	5
cg.B	18 301	339	182	342
ft.B	42 425	1286	1286	1286
is.C	11 622	726	125	1057
lu.A	1527	42	42	42
mg.B	3462	433	433	433
sp.A	1838	48	48	48
ua.A	1290	34	32	34

Table 4 describes the speedup of using the cache policy for the overall execution time. Table 5 shows the reduction rate of the overall cache miss count. These tables indicate that the cache policy can minimize the damage of cache invalidation via the cache footprint based on the working set size.

Table 4 Overall execution time for the three asymmetries

Capability balancing	Execution time (s)		Speedup
	FIFO	Cache	
Natural	574	548	4.5%
Dynamic	501	469	6.4%
Static	758	721	4.9%
Average			5.3%

Table 5 Overall cache miss count for the three asymmetries

Capability balancing	Cache miss		Reduction
	FIFO	Cache	
Natural	92 281 827	89 239 943	3.3%
Dynamic	133 638 374	125 278 680	6.3%
Static	117 890 129	112 491 834	4.6%
Average			4.7%

5 Related work

The concerns of process scheduling and load balancing have been the focus of much research. Process scheduling is used to fairly allocate CPU time to tasks in one processor and has been the subject of intensive research for several decades. The completely fair scheduler (CFS) introduced in Linux 2.6.23 has similar designs to weighted-fair queuing (WFQ) (Demers *et al*., 1989), and thus obtains a constant positive lag bound but an $O(N)$ negative bound. The recent distributed weighted round-robin (DWRR) (Li *et al*., 2009) algorithm has an edge over the existing fair scheduling algorithms, which are inaccurate or inefficient and non-scalable in multiprocessors, especially large scale multi-core processors. It achieves high efficiency and scalability with distributed thread queues and small additional overhead to the underlying scheduler.

Load balancing has been investigated in recent multiprocessor systems. Boneti *et al*. (2008) proposed a dynamic process scheduler for the Linux kernel that automatically and transparently balances high performance computing applications in accordance with their behavior. It provides a software-controlled prioritization mechanism that allows biasing processor resource allocation and reduces the application's execution time by combining the effects of load balance and high responsive scheduling. Saez *et al*. (2010) proposed, implemented, and evaluated CAMP, a comprehensive AMP scheduler, which provides both efficiency specialization and existing TLP (thread level parallelism) specialization. Efficiency specialization ensures that fast cores are used for CPU-intensive applications, which efficiently utilize expensive features of these cores, whereas slow cores would be used for memory-intensive applications, which utilize fast cores inefficiently. They also

proposed a new light-weight technique for discovering which threads utilize fast cores most efficiently. But, as mentioned by the authors, their approach uses a simple model that relies solely on the last level cache (LLC) and assumes a stable latency, and may fail to successfully schedule some tasks that are less CPU-intensive. Koufaty *et al*. (2010) proposed bias scheduling for heterogeneous systems with cores that have different microarchitectures and perform differently. They defined an application bias as the core type that best suits its resource needs. By dynamically monitoring application bias, the operating system is able to match threads to the core type that can maximize system throughput. They implemented the bias scheduler over the Linux scheduler in a real system that models microarchitectural differences accurately and found that it can significantly improve system performance, in proportion to the application bias diversity present in the workload. Hofmeyr *et al*. (2010) presented a load balancing technique designed specifically for parallel applications running on multi-core systems. Instead of balancing runqueue length, their algorithm balances tasks according to their speed metric among cores. Our approach is similar to theirs, except that we use CPU capability rather than speed as the load metric. CPU capability represents the available power of a processor and includes the speed metric presented by Hofmeyr *et al*. (2010). Unlike Hofmeyr *et al*. (2010), we identified the source of CPU capability asymmetry. Lakshminarayana *et al*. (2008) proposed and evaluated a task size aware scheduling algorithm and a critical section length aware scheduling algorithm in asymmetric multiprocessors. Unlike our approach, they required the modification of applications to distinguish longest or critical jobs. Jiang *et al*. (2011) explored the use of asymmetric last-level caches in a CMP platform and proposed ACCESS architecture where the OS scheduler is aware of asymmetry in the cache space across the cores. However, the approach is made available only on a special hardware like the 4-core Xeon 5160 chip.

The OS becomes larger and more complex to support new types of hardware and features. This situation gradually increases the OS noise and results in deteriorated performance in various domains, especially in the high performance computing area.

Tsafrir *et al*. (2005) attempted to quantify the effect of OS noise using a probabilistic approach. They also implemented a fixed time quantum (FTQ) like benchmark and instrumented a Linux kernel to log OS interrupts. They set OS timer ticks as the main source of OS noise and argued that a tickless kernel is a possible solution to the noise problem. Ferreira *et al*. (2008) examined the sensitivity of real-world, large-scale applications to a range of OS noise patterns using a kernel-based noise injection mechanism implemented in the Catamount lightweight kernel. They demonstrated how noise is generated, in terms of frequency and duration, and how this impact changes with the application scale. Radojkovic *et al*. (2008) analyzed the major sources of OS noise on a massive multithreading processor (Sun UltraSPARC T1) running Linux and Solaris. They showed that, the overhead introduced by the OS timer interrupt in Linux and Solaris depends on the particular core and hardware context in which the application is running. Scogland *et al*. (2008) studied the interactions between multi-core architectures and the networking subsystem. They recognized a remarkable amount of asymmetry in the effective capability of the different cores. However, they used a few symptoms to judge the asymmetry. In contrast, we make a decision for the asymmetry using the CPU capability metric. Scogland *et al*. (2008) proposed a novel management library called SyMMer (systems mapping manager) that monitors these interactions, dynamically manages the mapping of processes to processor cores, and improves application performance. In contrast, our approach is applicable to shared memory multiprocessing like OpenMP and MPI applications whereas SyMMer is limited to the latter.

6 Conclusions

Modern operating systems follow the symmetric multiprocessing (SMP) principle to fairly distribute tasks to multiprocessors. This approach works well when there is no difference among the CPU capabilities. But dynamic capability asymmetry by the OS noise from gigabit networking or static asymmetry by the advent of new asymmetry processors makes the OS scheduler work improperly.

In this paper, we have inspected the impact of CPU capability asymmetry on the OS kernel load

balancer. We define CPU capability as the new load metric and propose the asymmetry-aware user-level load balancer, capability balancer. This does not disturb the behavior of the kernel load balancer but coexists with it. We have shown the superiority of our balancing mechanism via the experiments using a synthetic program and NPB benchmarks.

Our study suggests that OS designers or developers have to consider CPU capability asymmetry when designing and implementing their systems.

As future work, we plan to study the dynamic load balance interval, which changes according to the degree of asymmetry, and include a new power-aware task selection policy in our capability balancer. We will also apply the capability balancing technique to non-uniform memory access (NUMA) systems.

Acknowledgements

The ICT at Seoul National University provided research facilities for this study.

References

Asanovic, K., Bodik, R., Demmel, J., Keaveny, T., Keutzer, K., Kubiatowicz, J., Morgan, N., Patterson, D., Sen, K., Wawrzynek, J., *et al*., 2009. A view of the parallel computing landscape. *Commun. ACM*, **52**(10):56-67. [doi:10.1145/1562764.1562783]

Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lsinski, T.A., Schreiber, R.S., *et al*., 1991. The NAS Parallel Benchmarks. *Int. J. Supercomput. Appl.*, **5**(3):63-73.

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., 2006. Operating system issues for petascale systems. *ACM SIGOPS Oper. Syst. Rev.*, **40**(2):29-33. [doi:10.1145/1131322.1131332]

Beckman, P., Iskra, K., Yoshii, K., Coghlan, S., Nataraj, A., 2008. Benchmarking the effects of operating system interference on extreme-scale parallel machines. *Cluster Comput.*, **11**(1):3-16. [doi:10.1007/s10586-007-0047-2]

Boneti, C., Gioiosa, R., Cazorla, F.J., Valero, M., 2008. A Dynamic Scheduler for Balancing HPC Applications. Proc. ACM/IEEE Conf. on Supercomputing, Article No. 41, p.1-12.

Bovet, D.P., Cesati, M., 2005. Understanding the Linux Kernel (3rd Ed.). O′Reilly, p.258-293.

Brodowski, D., 2005. Current Trend in Linux Kernel Power Management, linuxtag 2005. Available from http://www.free-it.de/archiv/talks_2005/paper-11017/paper-11017.pdf [Accessed on Apr. 15, 2012].

De, P., Kothari, R., Mann, V., 2007. Identifying Sources of Operating System Jitter Through Fine-Grained Kernel Instrumentation. Proc. IEEE Int. Conf. on Cluster Computing, p.331-340. [doi:10.1109/CLUSTR.2007.4629247]

Demers, A., Keshav, S., Shenker, S., 1989. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Comput. Commun. Rev.*, **19**(4):1-12. [doi:10.1145/75247.75248]

Ferreira, K., Brightwell, R., Bridges, P., 2008. Characterizing Application Sensitivity to OS Interference Using Kernel-Level Noise Injection. Proc. ACM/IEEE Conf. on Supercomputing, Article No. 19, p.1-12.

Gioiosa, R., Petrini, F., Davis, K., Lebaillif-Delamare, F., 2004. Analysis of System Overhead on Parallel Computers. 4th IEEE Int. Symp. on Signal Processing and Information Technology, p.387-390.

Gioiosa, R., McKee, S.A., Valero, M., 2010. Designing OS for HPC Applications: Scheduling. Proc. IEEE Int. Conf. on Cluster Computing, p.78-87. [doi:10.1109/CLUSTER.2010.16]

Hill, M.D., Marty, M.R., 2008. Amdahl's law in the multicore era. *IEEE Comput.*, **41**(7):33-38. [doi:10.1109/MC.2008.209]

Hoefler, T., Mehlan, T., Lumsdaine, A., Rehm, W., 2007. Netgauge: a Network Performance Measurement Framework. Proc. High Performance Computing and Communications, p.659-671.

Hofmeyr, S., Iancu, C., Blagojevíc, F., 2010. Load Balancing on Speed. Proc. 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, p.147-158. [doi:10.1145/1693453.1693475]

Jiang, X., Mishra, A., Zhao, L., Iyer, R., Fang, Z., Srinivasan, S., Makineni, S., Brett, P., Das, C.R., 2011. ACCESS: Smart Scheduling for Asymmetric Cache CMPs. Proc. IEEE 17th Int. Symp. on High Performance Computer Architecture, p.527-538. [doi:10.1109/HPCA.2011.5749757]

Koufaty, D., Reddy, D., Hahn, S., 2010. Bias Scheduling in Heterogeneous Multi-core Architectures. Proc. 5th European Conf. on Computer Systems, p.125-138. [doi:10.1145/1755913.1755928]

Kumar, R., Farkas, K.I., Jouppi, N.P., Ranganathan, P., Tullsen, D.M., 2003. Single-ISA Heterogeneous Multi-core Architectures: the Potential for Processor Power Reduction. Proc. 36th Annual IEEE/ACM Int. Symp. on Microarchitecture, p.81-92.

Kumar, R., Tullsen, D.M., Ranganathan, P., Jouppi, N.P., Farkas, K.I., 2004. Single-ISA Heterogeneous Multi-core Architectures for Multithreaded Workload Performance. Proc. 31st Annual Int. Symp. on Computer Architecture, p.64-75.

Lakshminarayana, N., Rao, S., Kim, H., 2008. Asymmetry Aware Scheduling Algorithms for Asymmetric Multiprocessors. Proc. 4th Annual Workshop on the Interaction Between Operating Systems and Computer Architecture.

Li, T., Baumberger, D., Hahn, S., 2009. Efficient and Scalable Multiprocessor Fair Scheduling Using Distributed Weighted Round-Robin. Proc. 14th ACM SIGPLAN

Symp. on Principles and Practice of Parallel Programming, p.65-74. [doi:10.1145/1504176.1504188]

Olsson, R., 2005. pktgen the Linux Packet Generator. Proc. Linux Symp., p.11-24.

Pallipadi, V., Starikovskiy, A., 2006. The Ondemand Governor. Proc. Linux Symp., p.223-238.

Petrini, F., Kerbyson, D.J., Pakin, S., 2003. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8192 Processors of ASCI Q. Proc. ACM/IEEE Conf. on Supercomputing, p.55-71. [doi:10.1145/1048935.1050204]

Radojkovic, P., Cakarevic, V., Verdu, J., Pajuelo, A., Gioiosa, R., Cazorla, F.J., Nemirovsky, M., Valero, M., 2008. Measuring Operating System Overhead on CMT Processors. Proc. 20th Int. Symp. on Computer Architecture and High Performance Computing, p.133-140. [doi:10.1109/SBAC-PAD.2008.19]

Saez, J.C., Prieto, M., Fedorova, A., Blagodurov, S., 2010. A Comprehensive Scheduler for Asymmetric Multicore Systems. Proc. 5th European Conf. on Computer Systems, p.139-152. [doi:10.1145/1755913.1755929]

Salim, J.H., 2001. Beyond Softnet. Proc. 5th Annual Linux Showcase and Conf., p.165-172.

Scogland, T., Balaji, P., Feng, W., Narayanaswamy, G., 2008. Asymmetric Interactions in Symmetric Multi-core Systems: Analysis, Enhancements and Evaluation. Proc. ACM/IEEE Conf. on Supercomputing, Article No. 17, p.1-12.

Tsafrir, D., Etsion, Y., Feitelson, D.G., Kirkpatrick, S., 2005. System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications. Proc. 19th Annual Int. Conf. on Supercomputing, p.303-312. [doi:10.1145/1088149.1088190]

Journals of Zhejiang University-SCIENCE (A/B/C) **website, http://www.zju.edu.cn/jzus**

New section "Articles (accepted manuscripts) in Press" has been available since 2011. The articles in press can also be commented by the readers. Each time an article is commented or cited by an ISI-indexed journal or proceeding, an e-mail notification will be sent automatically to the author(s). For each article, statistics such as downloads, clicks, citations, and comments are given in the contents of each issue