Dynamic threshold for imbalance assessment on load balancing for multicore systems

Ian K.T. Tan a,⁎, Ian Chai b, Poo Kuan Hoonga

a Faculty of Computing and Informatics, Multimedia University, Jalan Multimedia, Cyberjaya, 63100 Selangor, Malaysia
b Faculty of Engineering, Multimedia University, Jalan Multimedia, Cyberjaya, 63100 Selangor, Malaysia

A B S T R A C T

The introduction of multicore microprocessors has enabled smaller organizations to invest in high performance shared memory parallel systems. These systems ship with standard operating systems using preset thresholds for task imbalance assessment to activate load balancing. Unfortunately, this will unnecessarily trigger task migrations when the number of tasks is a few multiples of the number of processing cores. We illustrate this unnecessary task migration behavior through simulation and introduce a dynamic threshold for task imbalance assessment that is dependent on the number of tasks and the number of processing cores. This is as a replacement for the static threshold that is used by standard operating systems. With the dynamic threshold method, we are able to illustrate a performance gain of up to 17% on a synthetic benchmark and up to 25% gain using the Integer Sort Benchmark from the National Aeronautics and Space Administration (NASA) Advanced Supercomputing Parallel Benchmark Suite.

1. Introduction

Parallel processing in the form of affordable multicore systems are prevalent in today’s environment and this has enabled research and development organizations to conduct scientific engineering computation that was once the realm of established corporates or government organizations [1]. With this, it has enabled more sophisticated and better quality products being produced that are affordable to the general public. Supercomputing vendors, such Cray, Inc., have been offering solutions for departmental needs and has enabled small organizations to be able to afford departmental or even workgroup supercomputer class systems that employ multi or many core microprocessors. This segment of the market makes up a significant percentage of the overall High Performance Computing market space as per the latest report by International Data Corporation (IDC) [2]. This trend of using multicore microprocessors will continue for the foreseeable future with microprocessor manufacturers such as Intel and Advanced Micro Devices (AMD) continuing to introduce more processing cores per silicon die. Intel has even revealed 1000 cores on a single die [3] and AMD has announced their Bulldozer microprocessors [4] in September 2011. Although parallel processing hardware is widely available, small organizations would not have the specialized technical resources to develop these applications or optimize existing off the shelf scientific applications for their parallel processing platforms. Traditionally, optimization such as operating system kernel tuning, implementation of specialized operating systems or application parallelization, are done by highly specialized technical personnel employed or contracted. Fortunately for these small organizations, application code can easily be parallelized with tools such as OpenMP [5], and off-the-shelf commercial applications are already parallelized, such as ANSYS, MSC/NASTRAN and RiverFLO-2D.
However, affordable parallel processing hardware relies on standard operating systems that have generally evolved from catering for uni-processors with extensions for multiprocessors. Linux, which has gained inroads into the scientific computing environment, has only introduced multi-processor support as recently as 1996 [6] through the implementation of a global spinlock mechanism on a single global task run queue. The scalability of the multiprocessor support was improved in year 2000, using multiple task run queues per processor, where each task run queue will have their own task scheduling algorithm and the operating system will have global load balancing between the task run queues for multi-processor support.

With more processing cores being packed into an enterprise class server, limited effort has been focused on load balancing between the run queues. The mechanism employed by mainstream operating systems such as Microsoft Windows and Linux triggers load balancing when a processing core is idle or when the periodic load balancer detects task imbalance between the run queues [7,8]. The imbalance variance is a preset threshold value in the operating system kernel or a tunable parameter. In the case of the Linux operating system, this parameter is usually either at a variance of 25% or 33% [9].

In this paper, we proposed and implemented a runtime dynamic threshold for the imbalance computation. Our contributions are:

1. Dynamic threshold can significantly reduce the total number of task migrations whilst having no significant additional overheads or impact on the load balancing performance.
2. A synthetic benchmark that is computationally and memory intensive reveals that the dynamic threshold algorithm can improve the performance by up to 17%.
3. Standard benchmarks using both the C and FORTRAN programming languages with OpenMP parallelization constructs were studied to further emphasize the benefits of our dynamic threshold implementation. The throughputs of executing multiple copies of these benchmarks were compared and improvements of up to 25% can be observed.

This rest of this paper is organized as follows: Section 2 provides an introduction to operating system scheduling with an emphasis on the current implementation of multi-processor load balancing. In Section 3, we introduce our dynamic threshold for the load balancing algorithm. We proceed to illustrate, in Section 4, that the dynamic threshold algorithm reduces the number of total migrations and that the algorithm does not introduce detrimental performance overheads through scalability benchmarks. In Section 5, we conducted empirical studies using our synthetic mergesort benchmark, the NASA Advanced Supercomputing (NAS) Parallel Benchmark (NPB) Integer Sort benchmark and the NPB Lower–Upper benchmark. In Section 6, recent related work are discussed and we conclude in Section 7.

2. Multi-processor scheduling

Operating systems such as Microsoft Windows and Linux apply a two-phase scheduling for multiprocessors. The first phase provides task scheduling on a processing core where each processing core will have an independent task run queue associated with it. The second phase provides the load balancing to distribute tasks amongst the run queues. Task scheduling is an established field of research and has been well documented in books [10–14]. For multiprogramming scheduling (where the operating system has to schedule multiple tasks), algorithms range from the uncomplicated round robin algorithm, scheduling based on priority of the tasks, shortest time remaining task first scheduling and guaranteed scheduling to innovative lottery scheduling algorithms. Current research in this area concentrates on providing fair sharing, either between users or between tasks. Work done by Ingo Molnar and Con Kolivas [15–17] on the Linux kernel scheduling to provide fair sharing between tasks were motivated by the need to improve the user interactivity experience with Linux, while work by Wong et al. [18] strives to prevent starvation of tasks. To provide for multiprocessor support, load sharing [10,12] was a simple extension from uni-processor scheduling. Load sharing uses a single task run queue where the scheduler, when invoked, will select from and distributes the task for a processor to execute. A major drawback of sharing a single task run queue is race conditions. As the number of processing cores increases, the scalability will be affected. Advances were made by operating systems through the implementation of separate run queues for each processing core [10]. This leads to the need of global load balancing between the run queues. On the selection of tasks for redistribution by the load balancer, the concept of scheduling tasks, that are inter-related, into groups for maximizing inter-process communications, synchronization and cache usage is the most popular direction by far [19–21]. This is called gang scheduling [10].

The importance of gang scheduling becomes more obvious with the concept of threads, where a thread shares a common memory space with its sibling threads. A simple scenario is when a thread sends out a message to a sibling thread, it will block itself and wait. If the other receiving thread is on another processor and is not scheduled yet, there is latency involved. If the receiving thread is on the same processor, it is more likely to be scheduled next and hence the latency will be reduced. The introduction of multicore processors with sharing of on-chip resources such as cache memory and communications infrastructure has prompted further investigations into gang scheduling [22–24]. Gang scheduling focuses on the need for execution locality. To address over subscription of parallel applications on available processing cores, dynamic scheduling was proposed [10]. Dynamic scheduling requires cooperation between the operating system and the parallelized applications. The concept is that when there are $n$ processors available for a single application execution, it would be ideal if it can be parallelized into $n$ tasks. However, in cases where there are more than 1 parallel application running, by dynamically altering the
number of tasks in an application (depending on the number of processors available and number of peer processes), it will improve the performance through the reduction of context switching. With current processor clock speeds, the savings on context switching is easily negated by the effort of dynamically altering the level of concurrency of the applications.

Other work on dynamic scheduling [25] includes the introduction of a separate class of applications for high performance computing (HPC) applications. In short, their work attempts to ensure that parallel HPC applications utilize all available processing resources instead of grouping them. This is because HPC applications are usually written using Message Passing Interface (MPI) constructs and these separate tasks will not have shared resources between them. This is in contrast to gang scheduling.

Our work is motivated by the need to execute multiple shared memory parallel applications with the reduction of task migrations in order to improve execution locality. We propose to improve the execution locality through the operating systems load balancer as opposed to an implementation in hardware that was described in the article by Kavi et al. [26].

3. Dynamic thresholding for load balancer triggering

Dynamic load balancing is not a new idea and is a well researched area for distributed parallel processing. The current state of research in this area includes introducing new load balancing algorithms for the distributed systems interconnect topology [27], hierarchical load balancing architecture for very large systems [28] and decentralized load balancing for distributed systems [29]. On distributed parallel processing systems, the benefits gained from communications reduction is easily more than sufficient to buffer the overheads introduced by complex load balancing algorithms. However, on single multiprocessor system, load balancing between the processors has been kept as simple as possible as the granularity of the communications overheads does not warrant the introduction of complex load balancing algorithms. In a multiprocessor enabled operating system scheduler, there are generally two methods to trigger the load balancing function.

The first is when a processing core becomes idle and requests for tasks from other processing core run queues. When a processing core becomes idle, it will send an interrupt to request for the scheduler function, which will in turn attempt to balance the load by migrating tasks from other run queues to the idle processor run queue.

The second is a periodic function that will assess whether there is an imbalance between the active run queues. An imbalance is determined by the evaluation of whether the ratio of the tasks in the processing core run queues differ by a certain ratio. If it is deemed imbalanced, it will then attempt to conduct load balancing by migrating tasks from the processing core run queue with the higher number of tasks to the less busy processing core run queue. The imbalance value is a pre-determined constant (threshold) set in the operating systems kernel, i.e.

$$\left(\frac{t(p_i) - t(p_k)}{t(p_j)}\right) > \text{threshold} \Rightarrow \text{triggerlb}$$

where $p$ is the processing core, $i,j \in \forall(p)$ and $t(p)$ is the number of tasks in the processing core’s run queue.

Using Linux as a representation of current operating systems, the default kernel settings uses an imbalance threshold ratio of 25% [9] for the triggering of the load balancer. When the number of tasks in the system is less than four times the number of available processing cores, this imbalance ratio will trigger unnecessary migration of tasks which will not result in a better load balancing of the tasks. For example, with 7 tasks executing on a 4 core system, in a balanced environment, 3 of the processing cores will have 2 tasks in its run queue whilst one processing core will have 1 task in its run queue. Since the ratio of the number of tasks in that run queue that has 1 task against any of the other run queues will be 1, it will be greater than the threshold and hence will trigger the load balancer which will migrate the task to another run queue. This will again result in the same distribution of tasks in the system. Although the threshold is configurable on the kernel level, it is not practical to seek technical resources to recompile the kernel each time a new application wants to execute and reconfigure it again when an application finishes executing. We propose a dynamic threshold value to be computed at each instance that the imbalance ratio is to be assessed. Our dynamic threshold is based on the number of processing cores over the number of total tasks in the system. This is to prevent unnecessary migration that will result in the same overall load distribution on the system. We depict this as,

$$\frac{M}{\sum_{k=1}^{M} t(p_k)}$$

where $M$ is the number of available processing cores and $t(p_k)$ is the number of tasks in the processing core $k$’s run queue.

This results in the following formula:

$$\left(\frac{t(p_i) - t(p_k)}{t(p_j)}\right) > \frac{M}{\sum_{k=1}^{M} t(p_k)} \Rightarrow \text{triggerlb}$$

In order for the dynamic threshold to result in a profitable execution, the product of the frequency of the imbalance assessment resulting in the triggering of the load balance and the average cost of task migration for the stock operating system, should be larger than for our dynamic threshold method over the same period of time. Assuming that the average cost of task migration over a prolonged period of time would be approximately equal for both scenarios, we can simplify it as profitable when,
LinSched simulation on task migration frequency

4.1. LinSched simulation on task migration frequency

LinSched [33], originally developed at the University of North Carolina, is a user space application that hosts the Linux scheduling subsystem. It is able to accurately simulate the actual Linux scheduler with the objective of allowing an easier alternative of modifying the Linux kernel scheduler code in user space that allows for easier debugging than developing it in kernel space. It allows kernel developers to modify the scheduler code in the simulator and observe the experimental scheduling algorithms. The original simulator shares most of its code with the actual Linux scheduler code of kernel version 2.6.23.14 with some instrumentation points for data collection purposes. With this, it even allows a rather seamless port from the simulator to the actual Linux kernel. The LinSched subsystem [33] consists of three main modules, (i) the environment module, (ii) the simulation engine and (iii) the stimuli. The environment module includes the Linux source and header files in order to ensure that the simulation accurately represents the kernel scheduler. This is the main simulation subsystem and provides the necessary abstraction of the Linux kernel. The simulation engine provides the API for the stimuli program. It provides functions for initialization, controlling of the simulation scenarios and integrates with the Linux scheduling policies. Finally, the stimuli is a C program that uses the APIs provided by the simulation engine. It contains the simulation initialisation, tasks creations and calls to execute the simulation. The LinSched simulation application has been updated by Manomohan [36] of Google, Inc. where they used it to model the behavior of their custom Linux schedulers prior to implementation. It now models the Linux kernel scheduler version 2.6.35 [34] and also provides for improved task modelling and easier pre-defined multiprocessor system layout. The improved task modelling allows for tasks to have various degrees of CPU or I/O bound characteristics. We added instrumentation to the linux_linsched.c to log the creation of the tasks to include the task ID (in the sleep_run_init() function) and each time the tasks are scheduled, the CPU IDs and tasks IDs are noted (in the sleep_run_handle() function). This log file is used to process the total number of migrations between CPUs for each task.

Depending on the number of processing cores, our linsched.c script (stimuli program) creates tasks equaling one to four times the number of processing cores and simulated each run for 60 seconds (denoted by 60000 RUN_TICKS). Each task created was configured to be CPU bound (we used 99% CPU, 1% Sleep to model it). For our dynamic threshold algorithm, the Linux kernel uses weights instead of distinct number of tasks. Each normal task created (nice value 0) will be assigned a

\[
\omega_t \left( \frac{(t(p_l) - t(p_i))}{t(p_i)} \right) > \text{threshold} > \omega_{dt} \left( \frac{(t(p_l) - t(p_i))}{t(p_i)} \right) > \frac{M}{\sum_{k=1}^{s} t(p_k)}
\]

where \( \omega_t \) is the frequency of task migration in the stock operating system whilst \( \omega_{dt} \) is the frequency of task migration in our dynamic threshold method.

From Eq. (4), it is evident that to achieve a profitable usage of the dynamic threshold to trigger for load balancing, the following two aspects are important:

1. The frequency of load balancing triggering for the static threshold (stock) needs to be significantly higher than that of the dynamic threshold trigger.
2. The overheads introduced by \( \frac{M}{\sum_{k=1}^{s} t(p_k)} \) needs to be lower than the cost of task migration.

Our proposed dynamic threshold is beneficial for multiple applications that are well threaded, where the number of threads in the application is equal to the number of processing cores available on the system [18]. Our argument is that many engineering and scientific computing applications are threaded automatically to use the same number of processing cores that are available. Tools such as the OpenMP library provide for this and is used by commercial software such as MACROS and RiverFLO-2D.

Another important proponent to the profitable execution using our dynamic threshold is the cost of task migration. Tan et al. [30] measured the cost of task migrations of various sizes on symmetrical multiprocessing (SMP) systems, shared cache multicore systems and private cache multicore systems. From their measurement results, the cost of migrating large (in terms of memory usage) tasks is significant for SMP and private cache arrangement multicore processors. Lam et al. [31] also states that task migration should be minimized where possible.

However, concerns with any overheads introduced in a high performance computing environment is important as scientific and engineering applications typically execute for hours, if not days. As such, overheads will compound rapidly over the course of the application execution. Work done by Gu et al. [32] on introducing an efficient real-time operating system for non-shared memory supercomputer was also concerned with overheads in executing engineering applications.

4. Task migrations and overhead observations

In this section, we address the two important parameters that will affect the success of our dynamic threshold algorithm, namely the frequency of the migration and the overheads of implementing the algorithm. On the study of the migration frequency, we used the LinSched simulator [33,34] whilst for the observation of the overheads, we used the popular Linux kernel hackers scalability benchmark, Hackbench [35].

4.1. LinSched simulation on task migration frequency
weight of 1024. The imbalance assessment in the Linux kernel uses integer arithmetic (as this will consume fewer clock cycles than using floating point arithmetic) and is done using the following code (in sched_fair.c),

\[
100 \times \text{sds.max_load} \leq \text{sd->imbalance_pct} \times \text{sds.this_load}
\]

where \(\text{sds.max_load}\) is the total weight in the system, \(\text{imbalance_pct}\) is the imbalance percentage and \(\text{sds.this_load}\) is the total weight in the current run queue (or domain depending on the level on which the load balancing is acting).

The \(\text{imbalance_pct}\) is an integer value where 25% is presented using 125 and 33% is represented using 133. As such, our actual implementation in the Linux kernel to determine the current threshold value is as follows (for a 4 core system),

\[
\text{sd->imbalance_pct} = 100 + (100 \gg \text{sds.total_load} \gg 13);
\]

We use bit shift operators where we shift 10 bits to the left to equate the weight of 1024 with one task and a further 2 bits left for a 4 core system and an additional bit shift left to capture the lower bound of the multiples of processing cores. We simulated for (i) a quad core system, (ii) a dual socket quad core system (8 cores in total) and (iii) a quad socket quad core system (16 cores in total). For each simulation, we created tasks numbering 1–4 times the multiple of processing cores available and observed the total number of migrations.

The simulation runs (Fig. 1) indicate that when the number of processes is between 1 and 2 times the number of processing cores, the number of migration increases significantly. When the number of processes is an exact multiple of the number of processing cores, the imbalance assessment results in no load balancing as the tasks are perfectly balanced among the processing cores after the initial migration. The LinSched simulations show that there are significant migrations in the stock scheduler and this can be decreased by dynamically adjusting the load balancing threshold trigger.

We like to emphasize that we are not pursuing strict processing core affinity as our work allows for multicore system sharing among users and applications. The threshold value should also not be hardcoded into the kernel as the system is

![Fig. 1. Total migrations on a simulated (a) quad core system, (b) dual socket quad core system (c) quad socket quad core system.](image-url)
targeted for a shared multicore system, where the number of concurrent applications is undetermined at any given point in time. Hence, we proposed a dynamic threshold approach for triggering of the load balancing function.

4.2. Scalability and overheads benchmark using Hackbench

Through the LinSched simulations, we have established that we are able to reduce the migration by using a dynamic threshold. On the overheads introduced by equation \( t(\sum_{m} P_{m}) \), we measured the scalability of the overall implementation against the stock kernel. If the overheads are significant, then there will be a distinct difference between the stock kernel and our dynamic threshold implementation. We tested this on an actual kernel implementation of our dynamic threshold on two machines.

1. Intel Core 2 Duo Q9550 (Quad Core) @ 2.93 GHz with 4 GB main memory running Ubuntu 11.04 Desktop Edition.
2. Dual socket Intel Xeon E5335 (2xQuad Core) @ 2.00 GHz with 4 GB main memory running Ubuntu 11.04 Server Edition.

The dynamic threshold code was included and the systems were compiled using the default system settings with no further optimization. We used the popular scalability test Hackbench [35] which was created by Ingo Molnar, the main Linux kernel scheduler maintainer. Hackbench creates multiple tasks that communicate with each other via pipes. In our study, we ran Hackbench from 40 threads to 1000 threads in increments of 40. We took the average of 50 runs and our observations are shown in Fig. 2. Our results show that even with 1000 tasks in the system, the benchmark results show that there is insignificant difference between the stock and our dynamic threshold implementation.

5. Experimental results

With the knowledge that our criteria for profitable implementation where the frequency of load balancing action is reduced and that our dynamic threshold implementation does not suffer from significant overheads, we ran three sets of benchmarks. The first set being a synthetic benchmark created to simulate a memory intensive CPU bound application. The second is the NAS Parallel Benchmark (NPB) Integer Sort (IS) and finally the NAS Parallel Benchmark Lower–Upper computation (LU).

For each of our benchmarks, we executed multiple runs, similarly to Bilski and Winiecki [37] and Chen and Chu [38] whose work were to conduct performance measurements on multicore systems for parallel encryption and for image reconstruction on multicore systems respectively. Bilski and Winiecki [37] measured performance in the region of milliseconds and required a large set of encryptions as each encryption time varies significantly. As such, their work requires a large sampling size of 1024 sets. On the other hand, Chen and Chu [38] measured performance in the region of seconds and used a smaller set of runs (64 runs) to obtain the average times. Furthermore, their image reconstruction times does not vary as much. Similarly to our work, we will have variances in our performance measurement due to background processes in the Operating System. For our first benchmark we used a similar number of runs to obtain an average (100 runs), as per work done by Chen. For the second and third sets of benchmarks, we used the NAS Parallel Benchmark and in order to ensure that we obtain a truthful measurement, we took the average of 11 runs. This is the accepted number of runs for computer systems performance measurement recommended by Jain [39] which is also used by lmbench [40] for Operating Systems performance measurement.

![Fig. 2. Hackbench for quad core and dual quad core systems.](image-url)
5.1. Performance measurement of synthetic benchmark with CPU bound background processes

Our synthetic benchmark is based on a parallel mergesort algorithm where we only observed the performance during the parallel execution phase. The synthetic benchmark is parallelized into optimal tasks, where the number of parallel tasks is equal to the number of available processing cores. In order to simulate another concurrent application, we used the SysBench benchmark [41]. SysBench is a performance benchmarking tool designed for operating system benchmarking running databases under demanding loads. The SysBench tools provide for CPU bound performance testing using prime number computation. However, we simply used it as a CPU bound background process and specified the computation for prime numbers to execute for a longer period than the synthetic benchmark. We ran our optimally parallelized synthetic benchmarks with the SysBench application running in the background that is parallelized from 1 to 8 separate parallel tasks. Furthermore, we varied our synthetic benchmark to conduct the mergesort on 2,000,000–5,000,000 integers, which translates to an approximate memory usage of 2–5 MB. We took an average of 100 runs and our observations are depicted in Fig. 3.

When the number of SysBench tasks increases from 5 to 8, the dynamic threshold method generally delivered better performance, up to a gain of 17% for our synthetic benchmark. Overall, with larger problem sizes, the gain in performance of our synthetic benchmark becomes more significant. We can also observe that at perfect load balance situations where the number of tasks in the system is in multiples of the number of processing cores, both the stock and our dynamic threshold system deliver about the same level of performance.

It is also observed that for five tasks and seven tasks scenario, the performance gain is better as compared to the scenario with six tasks. This is due to the fact that in the stock Linux kernel, the migration strategy has a preference for load balancing between neighboring processing cores that share a common cache [31]. In the quad core microprocessor that we ran the benchmark on, the microprocessor has two caches where each cache is shared by two of the processing cores. For the six tasks scenario, the load balancer will have a preference to balance the two additional tasks between the two sets of cache and hence migration cost is minimized [30]. In the 5 tasks and seven tasks scenario, the stock load balancer will not be able to effectively balance between the two caches as there will always be an imbalance between them and hence the tasks will be migrated across caches. As such, our proposed solution will display a more significant performance gain.

5.2. Throughput measurement using NAS Parallel Benchmark Integer Sort (IS) with Small SMP System (B) class problem size

The NAS Parallel Benchmarks [42–44] provides a small set of benchmarks that are derived from actual engineering applications predominantly from the field of computational fluid dynamics. The benchmark is designed to evaluate parallel systems and it is available for various programming models such as large parallel systems using MPI, shared memory parallel systems using OpenMP, high performance FORTRAN and even a Java version. For our purpose, we used the NPB version 3.3 OpenMP model where we select a subset from it. The OpenMP model is selected as it provides a close reflection on actual situations in small research and development organizations that is not able to invest in highly specialized technical skill sets for MPI or High Performance FORTRAN. The NPB provides 11 different benchmarks within each benchmark suite which includes Fast Fourier Transforms and Embarrassingly Parallel benchmarks. From the 11, we selected the Integer Sort and the Lower–Upper Symmetric Gauss-Siedel benchmarks. The former is written in the C programming language whilst the latter is written in FORTRAN. For the IS benchmark, we compiled using GCC version 4.4.3 with full optimization (-O3) and the OpenMP library (-fopenmp and -lgomp). In order to specify the number of parallel tasks, we set the OMP_NUM_THREADS environment variable to reflect the number of processing cores. In this experiment, we used our Quad Core System (Intel Q9550 with 4 GB main memory) and hence the OMP_NUM_THREADS is specified to be 4. NPB provides benchmarks for various machine
classes where Class W and Class A are for workstation class machines; Class B and Class C are for server class machines and Class D onwards are for larger parallel machines. Our Quad Core system can be classified as a workstation class machine. As such we ran the IS benchmark using the Class A and Class B problem sizes. Class W is too small for this instance as each run completes in the sub-second space.

We took an average of 11 runs and from our observation (Fig. 4) for the small Class A problem size, the difference between the stock kernel and our dynamic threshold implementation is small; in the worst case scenario, our dynamic threshold method suffers from around 7% throughput degradation. This can be attributed to the overheads in re-evaluating the threshold value.

Through increasing the problem size from Class A to Class B, we noted that using the averaging of 11 runs, we are able to observe (Fig. 5) a throughput gain of around 25% for four concurrent IS Class B benchmark where each IS Class B benchmark is parallelized into four concurrent tasks. Fig. 6 uses the median value of the 11 runs. This result complements and confirms the throughput gain in using the dynamic threshold method.

5.3. Throughput measurement using NAS Parallel Benchmark Lower–Upper matrix (LU)

For the LU benchmark, which is a matrix computation, we compiled using gfortran which is part of GCC. We used the same options as for the IS compilation, which is with full optimization (-O3) and the OpenMP library (-fopenmp and -lgomp). Again we ran the benchmark with OMP_NUM_THREADS environment variable set to 4 on our Quad Core System (Intel Q9550 with 4 GB main memory). As the benchmark has larger memory requirements, we used the Class W problem size. Our observations are depicted in Fig. 7. We observe that when 2 or 3 concurrent LU Class W benchmarks are executing, the average throughput gain by the dynamic threshold method is approximately 10–12%.

However, with four concurrent LU Class W being executed, the memory consumption of the four concurrent benchmarks increase and impact the system’s cache memory usage. The cache memory is insufficient to store all the data necessary for

![Fig. 4. Average throughput time for concurrent NPB Class A IS benchmark.](image)

![Fig. 5. Average throughput time for concurrent NPB Class B IS benchmark.](image)
the benchmarks and would need to be swapped with the main memory whenever a process is scheduled. This happens frequently as these individual benchmark processes are scheduled using a round robin algorithm (they are of the same priority). This is termed as cache trashing. Although both the stock and the dynamic thresholding load balancer will be affected by this, the stock load balancer is already set to only trigger the load balancer when there is a difference of 33% whereas the dynamic thresholding load balancer will incur overheads to derive the same. With the cache trashing, any additional computation will also need to compete for space in the cache and will therefore be more significantly impacted as compared to when there are sufficient cache space for efficient execution.

6. Related work

From the load balancing perspective for performance improvement, we are only aware of the work by Hofmeyr et al. [45] where the approach is not to address cache utilization but instead focuses on asymmetric multicore environments, and a later work by Hofmeyr et al. [46] that built upon the earlier work [45].

In Hofmeyr et al. [45], they proposed instrumentation within the operating system to evaluate the execution performance of the various run queues and load balance them appropriately. The evaluation of run queue performance was then extended to the evaluation of tasks execution performance in Hofmeyr’s following work [46]. This latter work addresses a similar problem that we are addressing where the number of tasks is slightly more than the number of available processors in a symmetric multicore environment. Hofmeyr et al. [46] evaluates the execution performance of tasks and classifies them as being “ahead” or “behind” their peer tasks. The load balancer will then proceed to assign the “ahead” tasks to “slow” processors while the “behind” tasks are assigned to “fast” processors. The evaluation shows promising results and was conducted in the user space environment. However, their work differs from ours in that ours is an actual implementation in the kernel space where nuance tasks from a multi-programming environment is present and the actual overheads introduced by the dynamic load balancer algorithm is taken into consideration in the empirical studies.
7. Conclusion

With affordable high performance departmental supercomputers being acquired and shared by engineers or scientists, there is a need to ensure that the stock operating systems that ship with these departmental supercomputers do not unnecessarily trigger load balancing for the concurrent parallel applications.

Current commercial operating systems implement a static threshold value to trigger a load balancing function and does not perform well in situations where the system is shared by a few executing parallel applications. This is because the static threshold method will unnecessarily trigger the load balancing function when the system should be deemed as balanced.

In this paper, we proposed and implemented a dynamic threshold algorithm for inter run queue load balancing to improve the cache utilization whereas all other previous work were from the gang scheduling approach.

Through our simulations using a synthetic mergesort benchmark and the NAS Parallel Benchmarks with a background SysBench benchmark, we have illustrated that our proposed method:

1. reduces the total number of task migrations,
2. does not introduce significant additional overheads,
3. enables better performance for our mergesort based synthetic benchmark of up to 17%, and
4. produces better throughput for a subset of the NAS Parallel Benchmark of up to 25%.

The dynamic threshold method proposed and implemented is for executing computationally intensive applications on a shared multicore system. Applications such as Computational Fluid Dynamics will benefit from such an approach. Organizations with dedicated computational servers and technical expertise may also find this approach of interest. However, it is likely that performing tailored kernel tuning such as hard process affinity or a tuned static threshold will be more beneficial for dedicated systems.

References

[4] Nitta S. AMD confirmed to release new CPUs at CeBIT, probably bulldozer chips, February 2011 [online].


Ian K.T. Tan received his Master of Science in Parallel Computers and Computation from University of Warwick, United Kingdom and his Bachelor of Engineering in Information Systems Engineering from Imperial College London, United Kingdom in 1993 and 1992, respectively. His area of research is in operating systems process scheduling, network communications for distributed computing and computer security.

Ian Chai received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign, USA, his Master of Science in Computer Science and Bachelor of Science in Computer Science from University of Kansas, USA in the year 2000, 1990 and 1988 respectively. His research interest is in programming pedagogy, parallel programming and distributed computing.

Poo Kuan Hoong received his Ph.D. from the Department of Computer Science and Engineering at Nagoya Institute of Technology, Japan, his Master of Information Technology and Bachelor of Science from University Kebangsaan Malaysia, Malaysia in 2008, 2001 and 1997 respectively. His area of research interest includes peer-to-peer networks, parallel processing and distributed systems.