

An Adaptive Task-Core Ratio Load Balancing Strategy for Multi-core Processors

Ian K. T. Tan, *Member, IACSIT*, Chai Ian, and Poo Kuan Hoong

Abstract— With the proliferation of multi-core processors in servers, desktops, game consoles, mobile phones and a magnitude of other embedded devices; the need to ensure effective utilization of the processing cores becomes essential. This calls for research and development emphasis for a well engineered operating systems load balancer for these multi-core processors. In this paper, an adaptive load balancing strategy is presented. The adaptive load balancer will trigger tasks migration based on the tasks to processing core ratio, as well as when a processing core becomes idle. In our work, we utilize LinSched, a Linux operating system scheduler simulator, to analyze the number of task migrations. The Linux operating system is representative of the whole spectrum of computing as it is used in supercomputers, servers, desktops, mobile phones and embedded devices. Results from the simulation show that unnecessary task migrations were eliminated whilst maintaining the load balancing function effectively, as compared to the default strategy employed by the Linux operating system. The overheads introduced by the adaptive load balancer were measure through implementing it in a Linux kernel and measurements were made using the hackbench scalability test. The implementation proves to have negligible effect on the scalability and we can conclude that it does not introduce overheads. From our research, it shows that the adaptive load balancer provides a scalable solution for a lower and more consistent triggering of task migrations.

Index Terms— *operating system scheduling; load balancing; adaptive; task migration; multicore.*

I. INTRODUCTION

With the introduction of multi-core processors for servers and desktops, it was noted that a new operating system design is needed to fully appreciate and utilize the shared resources available on the processing cores [1]. Various aspects of shared resources require consideration; such as sharing of caches and also memory controllers within the processing cores. With most new microprocessors manufactured being multi-core processors; software developers have to exploit multi-threading programming techniques to achieve performance improvements. As such, the load balancer for multi-processing becomes key for multi-core processor design as features such as shared caches should be taken into consideration.

Manuscript received May 26, 2011; revised July 6, 2011.

Ian K. T. Tan is with Faculty of Information Technology, Multimedia University, Cyberjaya 63100 Selangor, Malaysia. (e-mail: ian@mmu.edu.my).

Ian Chai is with Faculty of Engineering, Multimedia University, Cyberjaya 63100 Selangor, Malaysia. (e-mail: ianchai@mmu.edu.my).

Poo Kuan Hoong is with Faculty of Information Technology, Multimedia University, Cyberjaya 63100 Selangor, Malaysia. (e-mail: khpoo@mmu.edu.my).

In maximizing multi-processing systems, load balancers are employed to ensure effective utilization of the processors. The two triggers for load balancing actions would be when a processing core becomes idle or when there is an imbalance detected between the processing cores during periodic load balancing checks. Such a technique is employed by the Linux operating system [2]. In the Linux load balancer, it will seek to balance the load if it detects a variance of 10%, 25% or 33% depending on the configuration. The different variance settings are dependent on the processor configuration of the multi-processor systems and the type of expected workload for the system.

With the price performance ratio of multi-core processors reducing rapidly as these processors become predominant; inexpensive dedicated systems such as personal supercomputers and embedded multi-core systems become widespread. In such systems, the applications that execute on such a system will likely be optimally threaded where the level of parallelism in the application will be equal to the number of processing cores available [3]. We argue that on such dedicated systems, there may be at most 2 or 3 such applications being executed concurrently. Hence the total number of processing tasks in the system will range between 1 to 4 multiples of the number of processing cores, in addition to a few other system (kernel) tasks.

In this condition, the design of the load balancer will then be crucial as the number of task migrations should be minimized in order to avoid expensive task migrations, especially across processing cores that do not share any caches. Lam et. al. [4] measured how cache affinity affects performance of memory intensive benchmarks and they have also showed that a fixed affinity (cache bound) improves performance significantly. However, this is not a practical solution as to obtain the significant performance improvement, the measurement for fixed affinity measured by Lam et. al. [4] requires a manually coded task to processor allocation and would imply that there will be no load balancing when certain tasks complete earlier than others.

Moreto et. al. [5] addresses the sensitivity of cache for application performance through the dynamic allocation of cache during run-time. In addressing the scientific computing workspace, where parallel applications are typically of the Single Instruction Multiple Data (SIMD) category, the overall performance of the application is determined by the slowest executing parallel thread. Their work involves additional hardware circuitry to allocate additional cache space for the slowest executing thread in order to improve the overall application performance.

Fedorova et. al. [6] approached the issue of the slowest thread by allocating more CPU time slice by the operating

system's scheduler instead of with the assistance of additional hardware circuitry. The net effect is similar to that of Moreto et. al. [5] in ensuring that the slowest thread is given more assistance and indirectly, this thread remains on the same cache for the longer period of executing time.

Hofmeyr et. al. [7] proposed an algorithm that they called *speed balancing*. Instead of the use of weights as in the native Linux load balancer, they substituted the computation for load balancing to be based on speed, where speed is defined as the CPU elapsed time divided by the actual wall clock time. It also introduces a balancer thread which will be periodically woken up and will pull tasks if the local core speed is greater than the average speed of all the cores in the system. The authors argued that the migration to a faster core will be able to effectively compensate for the migration overheads across caches. Their approach is however predominantly targeted at asymmetric processing cores.

Lam et. al. [4] and Tan et. al. [8] have published about the importance of taking cross cache migration into consideration where migration to another processing core that shares the same cache should be given preference whilst those that are on different chip should be given the least preference. In Fig. 1, the memory access denoted by α , will be faster compared to β , which in turn is faster than γ . In addition to memory access latency, Tan et. al. [8] have also measured the task migration performance impact when migration tasks of various sizes across caches.

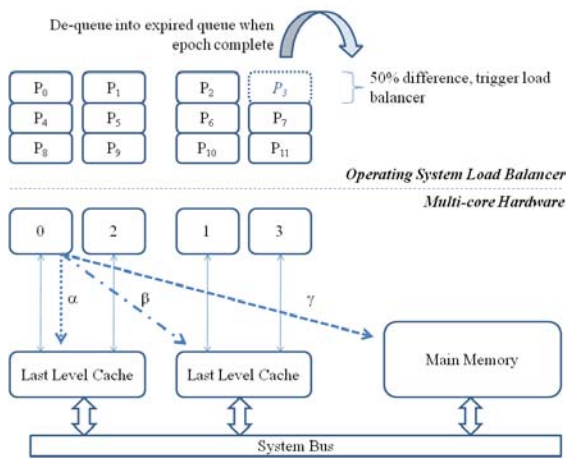


Fig. 1. A four processing cores system illustrating various memory latencies and load balancing triggering variance for 12 tasks on four processing cores.

In the work done by Lam et. al. [4], they utilized the Linux scheduling domain, which enables the Linux operating system to be processing core topology aware. The scheduling domain provides an indication to the operating system on how the hardware resources are shared and by extending this to be cache sharing aware, Lam et. al. [4] managed to obtain performance improvements. There have been attempts by Shi et. al. [9] to introduce a load balancer that utilizes the scheduling domain hierarchy of the Linux kernel scheduler, through the introduction of two threshold parameters. These parameters are not dissimilar to that already found in the native Linux load balancer. The parameters introduced are similar to the `migration_cost` variable in the native Linux scheduler and was computed

during boot up in the Linux kernel in version 2.6.22 and before.

The remaining part of this paper is organized as follows; Section 2 presents the static variance load balancers that are currently being used by general purpose operating systems such as Linux. An introduction to our proposed adaptive load balancer is in Section 3. Section 4 provides the details on the implementation in a scheduler simulator and brief discussion on the results. In Section 5, implementation in an actual Linux kernel with discussion on task migration results from executing a compute intensive benchmark. We provide a scalability test on our implementation in our conclusion for Section 6.

II. STATIC VARIANCE LOAD BALANCER

Current operating systems are designed in such a manner where each processing core is assigned a task queue (runqueue) and where a static variance load balancer is implemented; the variance between the numbers of tasks on the processing cores' runqueues is predetermined during system startup. When this preset variance is detected, the load balancer will trigger tasks migrations. Using the Linux operating system as a reference, it is generally set to 25% variance (denoted by the *imbalance* value of 125). Effectively, the load balancer will trigger tasks migrations when the ratio between two runqueues is greater than 5:4. In such a setting, when there are a low number of tasks, as typically will be on a dedicated system, it will unnecessarily trigger tasks migration activities. This is illustrated in Fig. 1 where, if there are 12 tasks on a four core system running concurrently and 1 task completes its allocated epoch (time slice in a multi-tasking system), the ratio between that runqueue and any others will be 3:2, or a 50% variance.

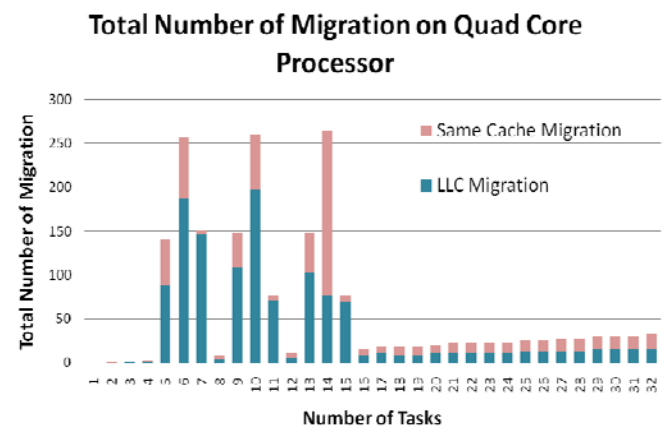


Fig. 2. Task migrations on a four core system for varying number of tasks; with indications of same cache and cross cache migrations.

Using the open source LinSched simulator [9] developed by Calandrino et. al., we simulated the execution of varying number of tasks over a period of 10,000 clock ticks. The data obtained by the simulator enables us to monitor the number of tasks migration and also enable us to determine whether the task migrations were across caches. Fig. 2 depicts our simulation results for the default load balancer as implemented in the Linux operating system kernel version 2.6.23 that implements the Completely Fair Share (CFS)

scheduler.

As indicated in the Fig. 2, when the tasks are evenly distributed among the four cores, minimal or no migration occurs but when there is just an imbalance of 1 task in the system, it triggers numerous tasks migrations. Beyond 16 tasks, the ratio of any additional tasks will not trigger migration as the ratio will be greater than 5:4. The migrations that are recorded are for the initial migration of the tasks by the load balancer to a less busy processor. Tasks in the Linux operating system are created initially on the first processor. It should also be noted that the number of cross cache migration consistently amounts to approximately 67% of the migration which indicates that the default Linux operating system load balancer gives no preference for same cache processing core migration [4]. An example of cross cache migration is illustrated in Fig. 3 as the migration from processing core 1 to processing core 2 (marked as ϕ in Fig. 3) where the migration core pair does not share the same last level cache. Same cache migration is illustrated as τ (in Fig. 3) where migration core pair (core 1 and core 3) shares the same last level cache.

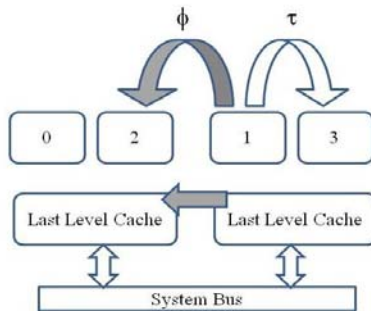


Fig. 3. Same cache migration and cross cache migration.

Instead of a predetermined value to trigger tasks migrations, we introduce an adaptive load balancer that is based on the task to processing core ratio.

III. ADAPTIVE LOAD BALANCER TASK MIGRATION

In the native Linux load balancer, the triggering of task migration is activated when the difference of the total weights of the tasks in the runqueue differs by a tunable Linux kernel parameter. This value is typically set to 125 (representing 125%) for most situations. It can be increased or decreased depending on the system administrator. The value is used when the operating system is comparing the current runqueue with the busiest runqueue of the overall system in the function `find_busiest_group()`. The function `find_busiest_group()` goes through a few checks prior to checking on the imbalance of the runqueues;

- The current runqueue is set to not do any load balancing.
- All other runqueues are not busy (e.g. no task running on them).
- The current runqueue is the busiest in the scheduling group.
- The current runqueue is busier than the average busyness of the entire scheduling domain group and hence it would not be suitable to pull tasks to it as it will then increase its busyness.

After which, the check on the imbalance will be made

through the following comparison;

$$100*sds.max_load <= \backslash sd->imbalance_pct*sds.this_load$$

The maximum load value of the runqueues of the scheduling domain (`sds.max_load`) is multiplied by 100 to enable integer arithmetic comparison to the right hand side of the code where the value 125 represents 125%. In our adaptive load balancer implementation, we dynamically change the value of `sd->imbalance_pct` to be based on the task to core ratio in the scheduling domain.

$$imbalance = 100 + \frac{100 \cdot nP}{mT} \quad (1)$$

where nP is the total number of processing cores in the scheduling domain and mT is the total number of threads running in the scheduling domain.

IV. LINSCHED IMPLEMENTATION

For our simulation we use an existing variable, `nr_threads` (not provided by LinSched but provided in the actual Linux kernel which we re-introduced into our LinSched simulator) to represent the number of tasks in the system and a fixed value for the number of processing cores.

Using integer arithmetic, the formula will effectively assign the value 150 to the `imbalance` variable when the ratio of tasks over processing core is between 2 and less than 3 to 1; the value 133 when the ratio is between 3 and less than 4 to 1; and the value 125 when the ratio is between 4 and less than 5 to 1. This is as compared to the static variance settings of the variable `imbalance` at 125 in the default load balancer.

We implemented this formula in the function `find_busiest_group()` in the main kernel scheduler source file, `sched.c`. The `imbalance` variable is represented by the code `sd->imbalance` and we implemented this recalculation of the imbalance variable just before the usage of it to compute whether there are any imbalance between the processing cores runqueues.

$$sd->imbalance_pct = \backslash 101 + ((100*4)/nr_threads);$$

We use 101 instead of 100 to avoid any rounding issues with the computation in order to ensure correct triggering of the task migrations.

A. Linsched Simulation

We executed two sets of simulations;

- a) Simulated 10,000 clock ticks with a fixed number of tasks ranging from 4 to 16 tasks and 25 tasks. We compare the migration using our proposed adaptive load balancer with the default static variance load balancer.
- b) Simulated 7,000 clock ticks with a fixed number of tasks ranging from 4 to 16 tasks and 25 tasks; then terminate all tasks on one processing core and continue the simulation for another 3,000 simulated clock ticks. We ran this for both the default static load balancer and our

proposed adaptive load balancer where our objective is to illustrate that the adaptive load balancer does trigger when there is a distinct imbalance in the processing cores' runqueues

B. LinSched Simulation Results

For the native Linux load balancer, the number of tasks migrations activation is minimal when the number of tasks is divisible by the number of processing cores or the number of tasks is greater than four times than number of processing cores. This is similar for the simulation (b) where $\frac{1}{4}$ of the tasks were terminated after 7,000 clock cycles. The total number of migrations is therefore reduced as there are fewer tasks in the system over 10,000 clock cycles. Fig. 4 depicts the results of (a) simulation of a fixed number of tasks over 10,000 clock ticks and (b) simulation to illustrate that tasks migrations are done efficiently using the adaptive load balance.

Our simulated results conclusively indicated that the proposed adaptive load balancer reduced migration effectively and when there is a distinct imbalance of tasks between the processing cores after the termination of all tasks on one processing core, the load balancer is triggered just to do the migration to load the idle processing core. This is indicated by the marginal increase of the migration count for the adaptive load balancer. The default load balancer registers a lower overall migration count as it effectively simulated set number of tasks for 7,000 clock ticks whilst the last 3,000 clock ticks the number of tasks has been reduced by the termination of the tasks on one processing core's runqueue.

The results from the simulation with the tasks termination also shows that although we started with a number of tasks that is a multiple of the number of processing cores; that is 4, 8, 12 and 16, the number of migration is higher. This is caused by the reduction of the tasks when they are terminated from one runqueue and hence migration will be triggered on the default load balancer.

However, in an actual system, there will be other tasks executing from time to time, which we will call nuance tasks. The simulation results do not provide us with the actual task migration scenario but is able to illustrate the effectiveness of the Adaptive Load Balancer. The next section will provide an insight into the actual task migration scenario on a Linux operating system.

V. LINUX KERNEL IMPLEMENTATION

We used version 2.6.32 of the Linux kernel as our base. There have been some minor changes from the version used by the LinSched simulator, and this includes the separation of the load balancer code into the `sched_fair.c` source file from `sched.c`. In addition, to reflect the actual mechanics of how the Linux kernel treats each task, we used the total weights instead of the total number of tasks. In our implementation, we worked on the default weight allocation to a normal task, which are 1,024. Hence we used the following code;

```
sd->imbalance_pct = 101 + \
(100 >> (sds.total_load >> 12));      (2)
```

where `sds.total_load` is the total weights of all the tasks in the scheduling domain.

We use the bit shift operator `>>` as it requires less clock cycles than using the division operator (bit shifting 12 is equivalent to division by 1,024).

A. Sysbench

For our work load, we used SysBench [11]. SysBench is a performance benchmarking tool designed for operating systems running databases under demanding loads. Within the SysBench tool, it has a CPU test mode that computes prime numbers up to a value specified by `--cpu-max-primess` using the number of threads specified by `--num-threads`.

We specified a variable number of threads from 1 to 20 and set the maximum prime number for the computation to 20,000. In order to minimize the nuance tasks, we ran our experiments under single user mode with no network drivers.

The result is illustrated in Fig. 5, where it is evident that the native Linux load balancer is exceptionally active when the number of tasks used for the prime number computation is between the number of tasks that are divisible by the number of processing cores.

When the number of tasks is divisible by number of cores or is greater than 4 times the number of cores (in this instance 16), the total tasks migrations activation is similar for both the native Linux implementation and the Adaptive Load Balancer, similarly to the LinSched simulation that was depicted in Fig. 5.

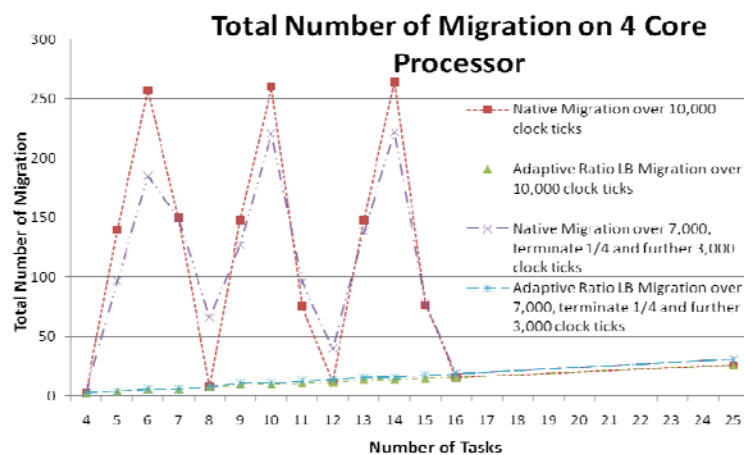


Fig. 4. Task migrations on a four core system for varying number of tasks; (i) for 10,000 clock cycles and (ii) for 7,000 clock cycles and a further 3,000 clock cycles after all tasks on one processing core are terminate.

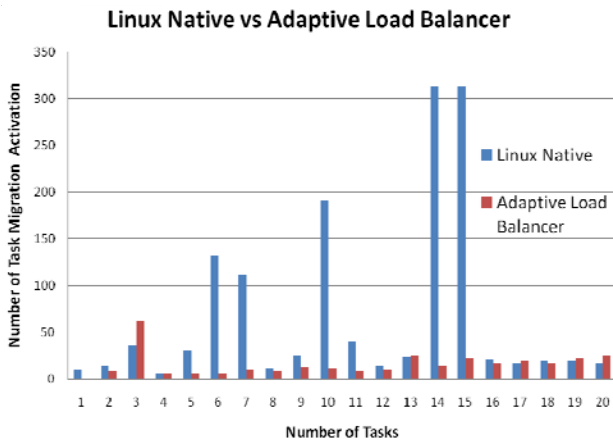


Fig. 5. Task migration activation on 4 core processor for native Linux load balancer and Adaptive Load Balancer when running SysBench computation for prime numbers.

Although we have been careful with our implementation in the Linux kernel as in the code provided in (2), one of the concerns of introducing additional code is the overheads and scalability of the system as a result of the addition. For this, we ran the Hackbench tool.

B. Hackbench

The Hackbench [11] tool is used to measure the overheads introduced in a modified Linux kernel. It simulates multiple piped connections between tasks and measured the time taken to pass tokens between tasks. We ran 40 to 1,000 tasks, incrementing them by 40 between tests. The results are shown in Fig. 6.

There is no observable difference between the native Linux native implementation and our Adaptive Load Balancer implementation which leads us to conclude that by introducing a re-computation of the *imbalance* variable, the overheads is negligible.

It is observed that the time variance between the Adaptive Load Balancer and the Linux native implementation ranges between -2% to 1.5% which is considered as not significant and overall it is on a downtrend (depicted by the dashed arrow in Fig. 6) as the number of tasks involved in the scalability test increases.

VI. CONCLUSION

It is evident in our proposed adaptive load balancer that takes into consideration the number of tasks in relation to the number of processing cores can be implemented to provide an effective load balancer. The results clearly show that the number of migration is reduced significantly as compared with the default scheduler.

Our adaptive load balancer will effectively represent the cache bound scheduler as described by Lam et. al. [4]. We achieve this with 2 significant differences;

- The cache affinity is achieved without manually programming the task placement, and
- Our proposed adaptive load balancer allows for load balancing when there is a significant imbalance.

Due consideration has to be taken into account for low overheads computation in kernel scheduling and the adaptive load balancer requires just a single line of integer arithmetic

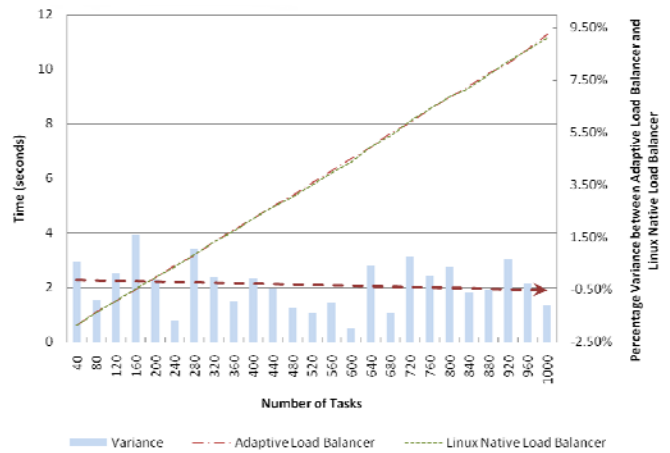


Fig. 6. Hackbench scalability test of up to 1,000 tasks for both native Linux load balancer and Adaptive Load Balancer.

computation.

With more dedicated computing, the number of tasks running on such systems will be in the low multiples of the number of available processing cores and short running nuance tasks by the system should not be a hindrance to performance by triggering unnecessary task migrations.

REFERENCES

- [1] A. Fedorova, C. Small, D. Nussbaum, M. Seltzer, "Chip Multithreading Systems Need a New Operating System Scheduler," in *Proc. 11th ACM SIGOPS European Workshop*, Leuven, Belgium, September, 2004,
- [2] W. Mauerer, *Professional Linux Kernel Architecture*, Wrox, USA, 2008, ch. 2, pp. 45-47.
- [3] C. S. Wong, I. K. T. Tan, R. D. Kumari, F. Wey, "Towards achieving fairness in the Linux scheduler," *ACM SIGOPS Operating Systems Review* vol. 42, no. 5, pp. 34-43, July, 2008.
- [4] J. W. Lam, I. K. T. Tan, B. L. Ong, C. K. Tan, "Effective Operating System Scheduling Domain Hierarchy for Core-Cache Awareness," in *Proc. IEEE Region Ten Conference (TENCON09)*, Singapore, November, 2009.
- [5] M. Moreto, F. J. Cazorla, R. Sakellariou, M. Valero, "Load balancing using dynamic cache allocation", in *Proc. 7th ACM international conference on Computing frontiers (CF '10)*, New York, NY, USA, 2010, pp. 153-164.
- [6] A. Fedorova, M. Seltzer, M. D. Smith, "Improving Performance Isolation on Chip Multiprocessors via an Operating System Scheduler," in *Proc. 16th International Conference on Parallel Architectures and Compilation Techniques (PACT 2007)*, Brasov, Romania, 2007, pp 25-38.
- [7] S. Hofmeyr, C. Iancu, F. Blagojevi, "Load balancing on speed", in *Proc. 15th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '10)*, New York, NY, USA, 2010, pp. 147-158.
- [8] I. K. T. Tan, I. Chai, K. H. Poo, "Pthreads Performance Characteristics on Shared Cache CMP, Private Cache CMP and SMP," in *Proc. 2nd International Conference on Computer Engineering and Applications (ICCEA2010)*, Bali, Indonesia, March 2010, pp. 186-191.
- [9] Q. Shi, T. Chen, W. Hu, C. Huang, "Load Balance Scheduling Algorithm for CMP Architecture," in *Proc. International Conference on Electronic Computer Technology*, Macau, China, 2009, pp. 396-400.
- [10] J. Calandrino, D. Baumberger, T. Li, J. Young, S. Hahn, "Linsched: The Linux Scheduler Simulator," in *Proc. 21st International Conference on Parallel and Distributed Computing and Communications Systems*, New Orleans, USA, September, 2008, pp. 171-176.
- [11] SysBench: a system performance benchmark. Available <http://sysbench.sourceforge.net/> (accessed March 2011)
- [12] Y. Zhang, "Improved Hackbench". Available <http://lkml.org/lkml/2008/1/4/18> (accessed March 2011)



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.

Prior to his current position as Lecturer at Multimedia University, Cyberjaya; he has worked in the area of microprocessor manufacturing, enterprise UNIX servers, storage management, software development to VoIP technologies. He is currently pursuing his Ph.D. at Faculty of Information Technology, Multimedia University. His area of research is in operating systems scheduling for multicore processors, data transfer optimization across wide area networks and text processing.

Mr Tan is a member is IACSIT, ACM, IEEE and an Associate of the City and Guilds Institute. He is Novell Certified Linux Administrator (NCLA), Novell Certified Linux Professional (NCLP)



Ian Chai received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign, Illinois, USA, his Master of Science in Computer Science and his Bachelor of Science in Computer Science from University of Kansas, USA in the year 2000, 1990 and 1988 respectively.

He currently holds the position of Senior Lecture in the Faculty of Engineering, Multimedia University, Cyberjaya, Malaysia and has been lecturing in

Multimedia University since 1999. Prior to that, he had worked as a research assistant and a teaching assistant at the University of Illinois and the University of Kansas, as a Research Programmer at the Forschungsinstitut für anwendungsorientierte Wissensverarbeitung in Ulm, Germany, and as a Technical Writer in the IT department of The Hartford (insurance company) in Connecticut, USA. He has co-authored a book, *Structuring Data, Building Algorithms* (Singapore: McGraw-Hill, 2009). His primary research is in the topic of how to structure documentation to teach people object-oriented frameworks they are unfamiliar with, but he also has an interest in parallel programming and distributed computing.

Dr Chai is a member of ACM.



Poo Kuan Hoong received his Ph.D. from the Department of Computer Science and Engineering at Nagoya Institute of Technology, Japan, his Bachelor of Science and Master of Information Technology from University Kebangsaan Malaysia, Malaysia in 2008, 2001 and 1997 respectively.

Since 2001, he has been lecturing at the Faculty of Information Technology, Multimedia University, Cyberjaya, Malaysia. Prior to joining Multimedia University, he lectured at several colleges in Kuala Lumpur. His area of research interest includes peer-to-peer networks and parallel and distributed systems. He also has in-depth knowledge of search engine optimization and internet marketing.

Dr Poo is a member of IEICE, IEEE and ACM. He is a Cisco Certified Network Associate Level I and II.