Regular Paper

Dynamic Estimation of Task Level Parallelism with Operating System Support

Luong Dinh Hung† and Shuichi Sakai†

The amount of task-level parallelism (TLP) in a runtime workload is useful information for determining the efficient usage of multiprocessors. This paper presents mechanisms for dynamically estimating the amount of TLP in runtime workloads. Modifications are made to the operating system (OS) to collect information about processor utilization and task activities, from which the TLP can be calculated. By effectively utilizing the time stamp counter (TSC) hardware, the task activities can be monitored with fine time resolution, which enables the TLP to be estimated with fine granularity. We implemented the mechanisms on a recent version of Linux. Evaluation results indicate that the mechanisms can estimate the TLP accurately for various workloads. The overheads imposed by the mechanisms are small.

1. Introduction

Multiprocessors are becoming increasingly widely used, and as a result there has been an increasing emphasis on thread-level and process-level parallelism. The term *task-level parallelism* (TLP) is used in this paper to refer to these kinds of parallelism.

The effectiveness of multiprocessors largely depends on the amount of TLP available in an execution workload. Multiprocessors are beneficial when the amount of TLP in the workload is large. By executing the constituent tasks in parallel across multiple processors, performance can be improved. Alternatively, multiprocessors with scaled voltage and frequency can provide lower power consumption than a uniprocessor with the same performance.

On the other hand, multiprocessors are ineffective for workloads with poor TLP. Since there are various paradigms of thread usage other than exploitation of parallelism [5], multithreaded applications with poor TLP are not uncommon [4),6]. Simply allocating many processors to execute such applications does not improve performance, but leads to power inefficiency. Some processors often remain idle while their tasks wait for data or synchronization from counterpart tasks being executed on other processors. The idle processors still require portions of power consumption; for example, some power is consumed in maintaining cache coherency [3], or by leakage. The leakage power already accounts for more than 30% of total power consumption in 90 nm technology [10], and increases rapidly with technology scaling [2]. When the TLP is poor, activating only a few processors for execution and shutting down the other processors definitely offers lower power consumption with insignificant performance degradation.

The amount of TLP in a workload is therefore important information for determining the appropriate number of processors that should be activated to execute the workload. If applications and their inputs are well understood in advance, which is a possible assumption for application-specific systems, the amount of TLP and consequently the power-efficient task scheduling can be determined statically [7]. However, such an assumption does not apply to general-purpose systems, where the applications and their inputs are unknown until runtime, so *dynamic* estimation of TLP is necessary.

There exist mechanisms that estimate TLP *statically* [4),9]. Hooks are added to the operating system (OS) [4], or multithreaded libraries [9] to collect information about tasks and processor utilization. TLP is then calculated off-line from the collected information. Since these mechanisms are intended for static analysis, the amount of collected data and the calculation overheads are not of concern. Indeed, the overheads are not reported in the abovementioned publications. However, when TLP is dynamically estimated for any online optimization, the factors become very important. It is thus unclear whether the existing mechanisms can be used for dynamic estimation of TLP.

This paper presents mechanisms for dynam-

† Graduate School of Information Science and Technology, The University of Tokyo

ically estimating the amount of TLP in runtime workloads. The mechanisms are realized through modifications to the OS to collect information about processor utilization and task activities, namely the state transition and transition timing of tasks. The TLP is calculated online from this information. By utilizing the time stamp counter (TSC) hardware, the process activities can be easily monitored in detail, resulting in estimation of the TLP with a fine granularity and with low overhead. Since no modification is required to applications and libraries, implementations of the mechanisms are transparent.

We implemented the mechanisms on Linux. Evaluation results indicate that our technique can estimate the TLP accurately for various workloads. The overheads imposed by the mechanisms are very small.

Section 2 describes the estimation targets, and defines of two kinds of TLP. Section 3 presents the estimation mechanisms. Implementation of the mechanisms in Linux is described in Section 4. Section 5 presents the evaluation results. Finally, Section 6 concludes the paper.

2. Task-Level Parallelism

This section first describes the kinds of tasks that are the estimation targets in this study. Since estimation of the TLP is essentially based on monitoring of tasks, we briefly cover task states and state transitions. We then define *inter-processor* TLP and *intra-processor* TLP, which are the two types of TLP to be estimated.

2.1 Estimation Targets

The mechanisms presented in this paper can estimate the parallelism of tasks that are the execution contexts *schedulable* by the OS. Eligible tasks are processes and kernel-level threads. Choosing them as the estimation targets allows optimizations to be made by the OS on the basis of the estimated parallelism. While the mechanisms does not estimate the parallelism of user-level threads, such parallelism to some extent can be inferred from the parallelism of kernel-level threads, or processes to which the user-level threads are mapped. Not estimating the parallelism of user-level threads therefore does not impose any significant limitations. Since no modification is required for applications and libraries, implementation of the mechanisms is transparent.

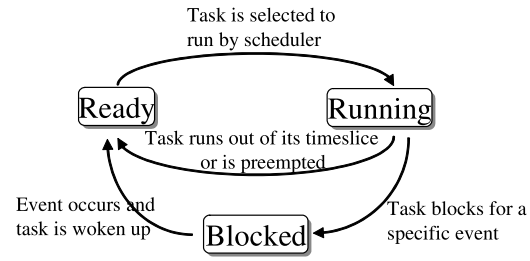

Fig. 1 Task state transitions.

2.2 Task States and State Transitions

Three of the typical states that a task can have are: *running*, *ready*, and *blocked*. **Figure 1** shows the state transitions of a task. When a task runs out of its timeslice, or is preempted by a high-priority task, the OS temporarily stops the running task and moves it to a runqueue (*running*→*ready* transition). The OS then selects a ready process from the runqueue and allows it to run (*ready*→*running* transition). When a running task needs to wait for an event, it will be pending in a waitqueue (*running*→*blocked* transition). The processor may now start to execute another task. When the event occurs, the blocked task is moved from the waitqueue to the runqueue (*blocked*→*ready* transition), making it eligible for later scheduling.

2.3 Inter-processor and Intra-processor Task-Level Parallelism

Inter-processor TLP, or *inter-TLP* for short, of a workload executed on a multiprocessor system is defined as the degree of concurrency with which the individual processors are busy during the execution of the workload. Those tasks executed on the same processor are treated as parts of a large cumulative task. Inter-TLP represents the parallelism among these cumulative tasks. Inter-TLP by this definition is identical with the TLP concept defined in a previous paper [4].

Intra-processor TLP, or *intra-TLP* for short, is defined as the TLP of those tasks that are executed on the same processor. TLP among the tasks in this case is implicitly buried inside the interleaving instruction streams from multiple tasks executed by the processor.

Inter-TLP and intra-TLP provide complementary views of TLP. The former represents the "global" view of TLP, while the latter represents TLP that is local to a processor. Poor inter-TLP signifies that the number of active processors should be reduced for power saving.

Conversely, large intra-TLP of tasks signifies that more processors should be activated to execute them. A heuristic algorithm can utilize the estimated results of both kinds of TLP to dynamically adjust the number of processors.

The mechanisms for estimating inter-TLP and intra-TLP are described in the following section.

3. Inter-TLP and Intra-TLP Estimation Mechanisms

Different mechanisms are needed in order to estimate Inter-TLP and intra-TLP. Estimation of inter-TLP can easily be achieved by monitoring processor utilization. Estimation of intra-TLP is more difficult, since we need to derive the parallelism information from the interleaving execution streams that are sequentially executed by a processor. We estimate intra-TLP by closely monitoring the activities of the individual tasks.

3.1 Inter-TLP Estimation Mechanism

Estimation of inter-TLP is achieved by forcing the OS to keep track of the durations that individual processors are busy. Inter-TLP is calculated from the degree of overlap among these durations.

The OS maintains execution time in a unit called a *time slot*. Each processor is required to keep information about whether it is busy or not in each time slot. The number of busy processors represents the degree of concurrency in the corresponding time slot. By collecting such data for all time slots over an observation period, the inter-TLP in that period can be determined by using Eq. (1).

$$TLP = \frac{\sum_{i=1}^{n} c_i i}{\sum_{i=1}^{n} c_i} \quad (1)$$

Here, c_i denotes the number of time slots in which exactly i processors are busy. The value of i ranges from 1 to n, where n is the number of active processors.

Figure 2 shows an example of how inter-TLP can be calculated. Figure 2 (a) shows the utilization of a multiprocessors system consisting of three processors P1, P2, and P3. The grey and white areas respectively indicate the periods the processor is busy and idle. In Fig. 2 (b), the utilization of each processor is quantized in time slot granularity. The number of processors busy in each slot is shown in Fig. 2 (c). In this example, the numbers of time slots in which exactly one, two, and three processors

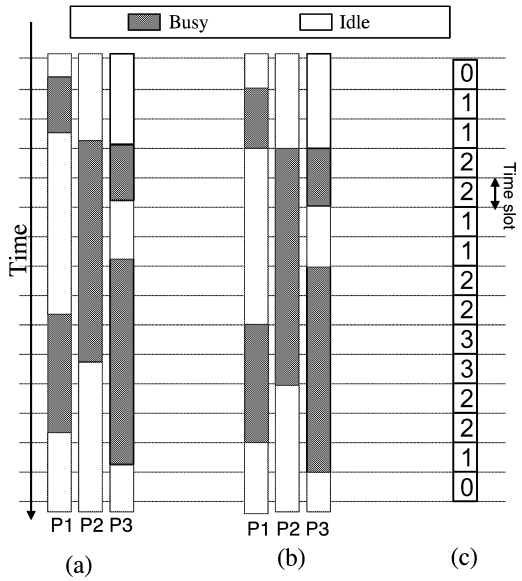

Fig. 2 Illustration of the inter-TLP estimation mechanism. (a) Utilization of three processors P1, P2, and P3. (b) Utilization quantized in time slots. (c) Concurrency for each time slot.

are busy are respectively 5, 6, and 2. Following Eq. (1), the inter-TLP for this example is $(5 * 1 + 6 * 2 + 2 * 3)/(5 + 6 + 2) = 1.77$.

It is clear from Fig. 2 that the accuracy of inter-TLP estimation depends on the size of the time slot. Using a fine time slot improves the accuracy at the expense of increased estimation overhead. Such a trade-off will be quantified in the evaluation.

3.2 Intra-TLP Estimation Mechanism

To estimate intra-TLP, the OS keep track of state transitions of tasks being executed on the same processor. The blocked, ready, and running durations of each task are identified. The collected sequence of durations represents the *real* trace of the corresponding task.

An important observation is made here. The existence of ready durations is merely due to a shortage of processors, rather to any true data dependency among tasks. Removing the ready durations from the real traces allows us to simulate in which the case the same tasks are executed on a multiprocessor system with an unlimited number of processors.

Removing ready durations from a real trace creates a new trace consisting of only blocked and running durations. The new trace is called an *imaginary* trace. Whereas there is no overlap among running durations in the real traces of tasks executed on the same processor, the

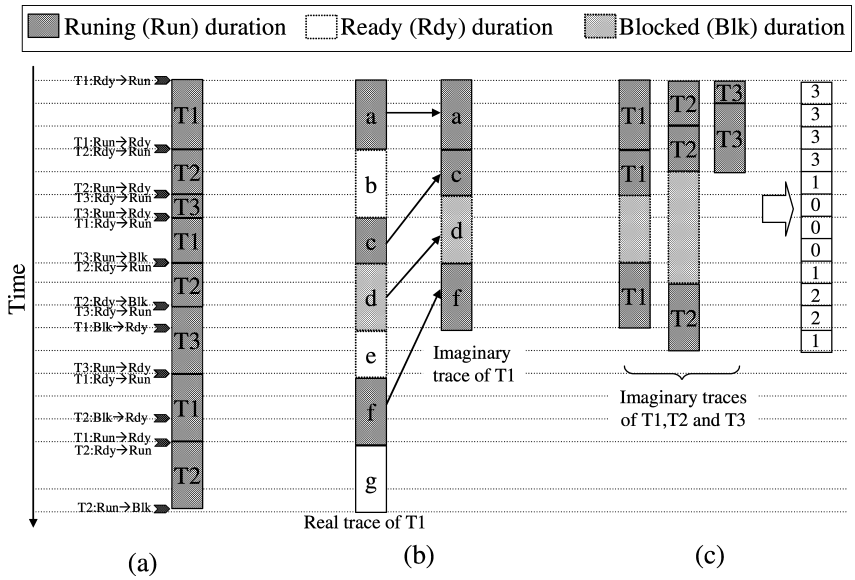

Fig. 3 Illustration of the intra-TLP estimation mechanism. (a) Three tasks, T1, T2, and T3 being executed on the same processor. (b) Real trace and imaginary trace of T1. (c) Imaginary traces of all tasks and the concurrency for each time slot.

running durations in their imaginary traces can overlap. The resulting overlap represents the intra-TLP.

After the imaginary traces have been generated, the degree of overlap among running durations of the imaginary traces is derived. The intra-TLP is then calculated on the basis of Eq. (1). The term c_i in this case represents the number of slots having i tasks in the running state indicated in the imaginary traces.

Figure 3 shows an example of how intra-TLP is estimated. Figure 3 (a) shows three tasks, T1, T2, and T3 being executed on the same processor. The state transitions of the tasks are shown on the timeline. Figure 3 (b) focuses on the execution of task T1. The real trace and imaginary trace, as well as the generation of the imaginary trace from the real trace, are indicated. Figure 3 (c) shows the imaginary traces of all tasks. Following Eq. (1), the values of c_1, c_2, and c_3 in Fig. 3 (c) are respectively 2, 2, and 4. The intra-TLP in this case is $(2 * 1 + 2 * 2 + 4 * 3)/(2 + 2 + 4) = 2.25$.

Comparison with another mechanism. One may argue for a simpler mechanism that estimates the intra-TLP on the basis of the number of ready tasks in the runqueue. However, such a mechanism does not produce a correct TLP. The reason may be explained as follows. The OS usually contains numerous low-priority *daemons* to handle miscellaneous housekeeping jobs. These daemons wake up frequently, run briefly and then go back to sleep. If the OS executes high-priority task(s), these daemons may wait for a long time in the runqueue for their turn to execute. Simply counting the number of tasks in the runqueue in such a case overestimates the TLP. We confirmed that the TLP of a DVD playback application estimated by this mechanism is more than twice as large as the actual TLP.

The TLP can be estimated by not counting those daemons. To provide responsiveness, the OS tends to reduce the priority of computation-intensive tasks. We therefore cannot filter out the daemons on the basis of priority alone, since the daemon tasks and computation-intensive tasks may have the same priority levels. The filtering must also take into account the processor occupation time. How to maintain such information and how to properly set the threshold values are not straightforward.

In the proposed intra-TLP estimation mechanism, since the lengths of the running durations are used in the calculation, these daemons, which typically have short run time, have little effect on the estimated TLP.

4. Implementation

We implement the proposed mechanisms on

Linux. To Linux, there is no concept of kernel-level thread; Linux implements all threads as normal processes [8]. We therefore just need to consider TLP estimation for normal processes in Linux. However, the implementation can easily be extended to support kernel-level threads in other OSs.

4.1 Implementation of Inter-TLP Estimation

The OS allocates an array shared by all processors. Each element of the array corresponds to a time slot in the observation period. If a processor is busy in a time slot, the value of the corresponding array element is increased by one. The update of the array is handled by a specific code added to a periodic timer (called every one millisecond), and also to the context switch routine. The code updates the array for those time slots since the time of its last execution until the current time slot.

The number of non-zero elements, and the sum of the values of the array elements, collected at the end of the period, respectively represent the denominator and the numerator in Eq. (1). The resulting inter-TLP is output and the array is reset at the end of the period to be ready for the next observation period.

4.2 Implementation of Intra-TLP Estimation

Figure 4 illustrates main modifications made to the OS in order to estimate intra-TLP. In Linux, process-related information is kept in instances of *task* data structure, with one instance per process. We newly add the variables *blocked_ts*, *ready_ts*, and *img_trace_length* to the *task* structure. Timestamps indicating that a process enters or exits the blocked state are respectively recorded in *blocked_ts* and *ready_ts* by the *state_tran* function in Fig. 4. *img_trace_len* represents the length of the process's imaginary trace. The length of the blocked duration, which is equivalent to *ready_ts*−*blocked_ts*, is added to *img_trace_len* when the process resumes its execution after having been blocked.

A bit array is maintained for each processor (the *busy* variable in the *cpu* struct in Fig. 4). If a process is in running state during a time slot, the element of the array indexed by the current value of *img_trace_len* of the process, is set to 1. The value of *img_trace_len* is then incremented by 1. Updates of the array and *img_trace_len* are incrementally done in the *update_slot* function in Fig. 4, which is called from an interrupt timer and the context switch routine.

```
struct task{ //per-task info
  //newly added variables
  int blocked_ts;
  int ready_ts;
  int img_trace_len;
  .....
}
struct cpu{ //per-cpu info
  //newly added variable
  bit busy[NSLOT];
  .....
}
//change the state of a task
func state_tran(task t, int newstate){
  if(newstate == BLOCKED)
    t.blocked_ts = now;
  else if (newstate == READY)
    t.ready_ts = now;
}
// update cpu usage in recent slots
func update_slot(task t){
  while(t.slot < now) {
    thiscpu.busy[t.slot%NSLOT] = 1;
    t.slot++;
  }
}
//context switch function
func ctx_sw(task prev, task next){
  update_slot(prev);
  next.img_trace_len =
       next.ready_ts - next.blocked_ts;
  next.ready_ts = next.blocked_ts = now;
  ......
}
//timer called every one millisecond
func timer(){
  //cur: currently executed task
  update_slot(cur);
  ......
}
```

Fig. 4 Pseudo code indicating the main modifications made to the OS in order to estimate intra-TLP.

The number of elements of the bit array whose bits have been set, collected at the end of the observation period, is the value of the denominator in Eq. (1). The numerator in Eq. (1) is calculated by summing up the lengths of all running durations of the processes over the period. At the end of the period, the inter-TLP is computed and output. The bit arrays and the *img_trace_len* of each process are reset accordingly.

Since the estimations of inter-TLP and intra-TLP are independent of each other, the OS can support simultaneous estimations of both kinds of TLP. The estimation accuracy is unchanged and the estimation overheads are additive.

4.3 Timekeeping Using a Time Stamp Counter

The size of the time slot affects the accuracy of the estimated TLP. A small time slot allows estimation of the TLP with fine granularity. However, if an existing time management facility in the OS (e.g., the *gettimeofdate* system call) is used for high-resolution time management in our estimation mechanisms, frequent timing requests unavoidably incur large overheads.

Our TLP estimation mechanisms instead manages timing by utilizing the time stamp counter (TSC), which is provided in most recent processors. TSC is a counter whose value is incremented by 1 in every clock cycle. Each processor in the multiprocessor system has its own TSC, and these counters are synchronized at boot time. Access to the TSC is simply a matter of reading an on-chip register. A desired timing resolution is easily obtained by reading the TSC and shifting the result appropriately. For instance, for a 1 GHz processor, reading the TSC and shifting the result to the right by 15 bits achieves a time resolution of 32 μs. The overhead of maintaining time in this manner is very small.

5. Evaluation

5.1 Evaluation Methodology

Our evaluation machine is a two-way multiprocessor system with 1.8 GHz Athlon processors and 1 GB of main memory. The machine runs a recent Linux OS (version 2.6.5). The OS is modified to incorporate the TLP estimation functions described in Section 4.

Two types of workloads are used for evaluation. The first type of workload consists of independent tasks from various single-threaded applications. The second type of workload consists of multithreaded applications. **Table 1** shows a list of benchmarks along with brief descriptions.

When we evaluate the inter-processor TLP, both processors are activated. When we evaluate intra-TLP, we activate only one processor. The effectiveness of the intra-TLP estimation mechanism can be evaluated on an uniprocessor without losing generality. The impacts of varying the size of the time slot on the accuracy and overheads of the proposed mechanisms are studied.

The overheads incurred by the TLP estimation mechanisms are defined as the increases in the execution times of the chosen workloads executed on the modified OS, compared with the execution times of the same workloads executed on the original OS. Applications from SPECINT benchmarks are chosen for overhead evaluations.

Table 1 Evaluation benchmarks.

Benchmark	Description
Single-threaded benchmarks	
mcf, *twolf*, *bzip2*	Integer applications in SPECCPU 2000
Multithreaded benchmarks	
mencoder	Video encoder software
mozilla	Web browser
dvdplayer	DVD playback
openssl	Cryptography toolkits
pigimp	Benchmark for GIMP

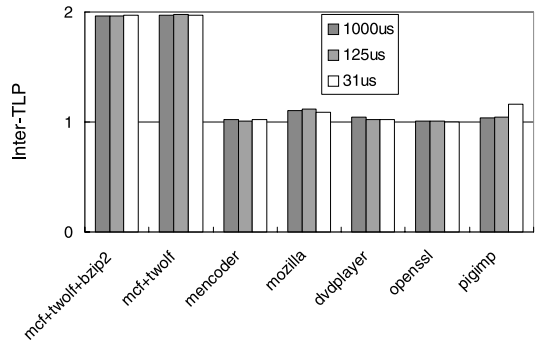

Fig. 5 Estimated inter-TLP for three different sizes of time slot.

5.2 Results

5.2.1 Inter-TLP Estimation Results

Figure 5 shows the estimated inter-TLP for several workloads. The size of time slot is varied among 1,000 μs, 125 μs, and 31 μs.

The leftmost two workloads consist of multiple computation-intensive, single-threaded applications executed concurrently. The real TLP is expected to be roughly equal to the number of applications in the workloads. The estimated inter-TLP of the *mcf+twolf* workload, i.e., approximately 2, is reasonable. However, since the number of applications in the *mcf+twolf+bzip2* workload is larger than the number of available processors, the inter-TLP in this case is estimated as 2, not as 3.

For those multithreaded applications, *mencoder*, *dvdplayer*, and *openssl* exhibit very limited TLP, confirming the fact that many applications are multithreaded for purposes other than to exploit parallelism [5]. *mozilla* exhibits slightly higher TLP than other multithreaded applications.

Table 2 Overheads of inter-TLP estimation for three different sizes of time slot.

Size of time slot (μs)	Overhead (%)
1,000	0.04
125	0.12
31	0.23

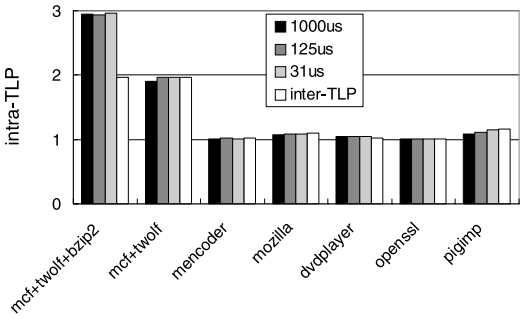

Fig. 6 Intra-TLP estimates for three different time resolutions.

Table 3 Overheads of the intra-TLP estimation at different sizes of time slot.

Size of time slot (μs)	Overhead (%)
1,000	0.11
125	0.21
31	0.35

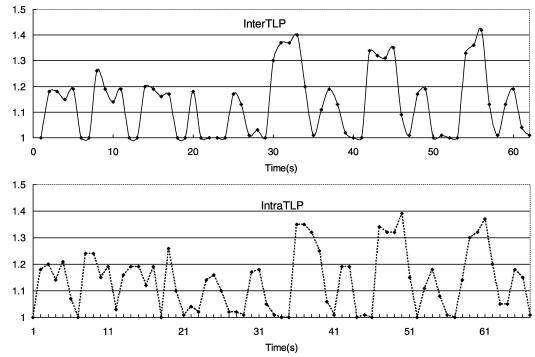

Fig. 7 Estimated inter-TLP, intra-TLP of *pigimp* vary with time.

For most workloads, the estimated TLP changes very little when the size of time slot is varied. One exception is *pigimp*. Since *pigimp* consists of communication processes that exhibit parallelism with fine granularity, the accuracy of inter-TLP estimation improves with a small time slot.

Table 2 shows the overhead of the inter-TLP estimation mechanism as the size of the time slot is varied. The overhead increases almost linearly with a reduction in the size of the time slot. The overhead is fairly low.

5.2.2 Intra-TLP Estimation Results

Figure 6 shows the estimated intra-TLP. The size of the time slot is also varied among 1,000 μs, 125 μs, and 31 μs. For reference, the inter-TLP estimated with a time slot of 31 μs, derived from the results in Section 5.2.1, is also shown in Fig. 6.

For a workload consisting of multiple single-threaded applications, the estimated intra-TLP of a workload is almost equal to the number of applications in the workload. It is noteworthy that while the inter-TLP for *mcf+twolf+bzip2* is estimated as 2, the intra-TLP is estimated as 3, which correctly matches the real TLP of the workload. Since the TLP extracted from concurrent execution of independent tasks represents the most common type of TLP found in practice, the ability of the mechanism to correctly estimate the TLP is encouraging.

For multithreaded applications, the estimated intra-TLP closely matches the inter-TLP. Since those multithreaded applications exhibit poor TLP, the estimated inter-TLP represents the real TLP of these applications. The estimate of intra-TLP is therefore accurate. A smaller time slot improves the accuracy of the estimate, as indicated by *pigimp*.

Table 3 shows the overhead of the intra-TLP estimation mechanism as the size of the time slot is varied. Since more computation is involved in estimation of intra-TLP, the overhead of intra-TLP estimation is larger than that of inter-TLP estimation. Nevertheless, the overhead is still fairly small (e.g., up to 0.35%).

The upper and lower area of **Fig. 7** respectively show the variation over time of estimated inter-TLP and intra-TLP for *pigimp*. Since the multiprocessor machine used in inter-TLP estimation provides higher performance than the single-processor used in intra-TLP estimation, the time in the upper graph is stretched to match the time in the lower graph. The estimated intra-TLP closely matches the inter-TLP. The proposed mechanisms work effectively for applications in which TLP varies with time.

6. Conclusions

This paper has presented two mechanisms for dynamically estimating the amount of task level parallelism (TLP) in runtime workloads. The first estimates the TLP among tasks executed on different processors of a multiprocessor system, and the second estimates the TLP among

tasks executed on the same processor. Appropriate modifications are added to the OS to collect information about processor utilization and task activities, on the basis of which the TLP is deduced. Utilization of the time stamp counter (TSC) allows such information to be collected with high time resolution and with low overhead.

The mechanisms are implemented on a recent version of Linux. The results indicate that the proposed mechanisms estimate the TLP accurately for workloads consisting of multiple independent processes as well as the workloads of multithreaded applications. The overheads imposed by the mechanisms are negligible.

The efficacy of dynamic estimation of TLP has been demonstrated in this work. We are working toward evaluating the effectiveness of power reduction through dynamic adaptation of the number of processors based on the estimated TLP on a real multiprocessor system. The ability to adaptively shutdown processors in order to save power on MPCore multiprocessor systems [1] makes them valuable platforms for future experiments.

Acknowledgments This research is partially supported by Grants-in-Aid for Fundamental Scientific Research B(2) #13480077 and B(2) #16300013 from Japan's Ministry of Education, Culture, Sports, Science and Technology Japan, and from the Semiconductor Technology Academic Research Center (STARC) Japan, the CREST project of Japan Science and Technology Corporation, and the 21st century COE project of Japan Society for the Promotion of Science.

References

1) ARM: http://www.arm.com/products/CPUs/MPCoreMultiprocessor.html.
2) Borkar, S.: Design Challenges for Technology Scaling, *IEEE Micro*, Vol.19, No.4, pp.23–29 (1999).
3) Ekman, M., Dahlgren, F. and Stenstrom, P.: TLB and Snoop Energy-Reduction using Virtual Caches in Low-Power Chip-Multiprocessors, *Proc. 2002 International Symposium on Low Power Electronics and Design*, pp.243–246 (2002).
4) Flautner, K., Uhlig, R., Reinhardt, S. and Mudge, T.: Thread-Level Parallelism of Desktop Applications, *Proc. Workshop on Multithreaded Execution, Architecture and Compilation* (*MTEAC 2000*) (2000).
5) Hauser, C., Jacobi, C., Theimer, M., Welch, B. and Weiser, M.: Using Threads in Interactive Systems: A Case Study, *Proc. 14th Symposium on Operating Systems Principles*, pp.94–105 (1993).
6) Lee, D.C., Crowley, P.J., Baer, J.-L., Anderson, T.E. and Bershad, B.N.: Characteristics of Desktop Applications on Windows NT, *Proc. 25th International Symposium on Computer Architecture*, pp.3–14 (1998).
7) Li, J. and Martinez, J.F.: Power-Performance Implications of Thread-level Parallelism on Chip Multiprocessors, *Proc. 2005 IEEE International Symposium Performance Analysis of Systems and Software*, pp.124–134 (2005).
8) Love, R.: Linux Kernel Development, Sams Publishing (2004).
9) Lundberg, L.: Predicting and Bounding the Speedup of Multithreaded Solaris Programs, *Journal of Parallel and Distributed Computing*, Vol.57, No.3, pp.322–333 (June 1999).
10) Schutz, J. and Webb, C.: A Scalable X86 CPU Design for 90 nm Process, *Proc. IEEE International Symposium on Solid-State Circuits Conference*, pp.62–63 (2004).

(Received September 29, 2005)
(Accepted February 1, 2006)

Luong Dinh Hung is currently a Ph.D. student in Information and Communication Engineering in The University of Tokyo. He received the M.E. degree in Information and Communication Engineering from The University of Tokyo in 2004. His research interests are in microprocessor architecture and circuit techniques for low-power consumption and soft-error tolerance.

Shuichi Sakai received the B.S., M.E. degree and D.E. from The University of Tokyo in 1981, 1983 and 1986 respectively. He worked in Electrotechnical Laboratory Japan from 1986 to 1990. From 1991 to 1992, he became a visiting scientist in Laboratory for Computer Science, Massachusetts Institute of Technology. From 1993 to 1996, he was a chief at Massively Parallel Architecture Laboratory in Real World Computing Partnership. He became an Associate Professor in University of Tsukuba in 1996 and came to The University of Tokyo in 1998 as an Associate Professor. From 2001, he has been a Professor in Graduate School of Information Science and Technology in The University of Tokyo. His research interests include dependable computer systems, microprocessor architecture, compiler, parallel computing, and multimedia processing. He wrote several books on logic circuits and computer architecture, including "Introduction to Logic Circuits" and "Computer Architecture with Illustrated Explanation". He is a member of IPSJ, IEEE, ACM, IEICE, and JSAI.