Dynamic Estimation of Task Level Parallelism with Operating System Support

Luong D. Hung and Shuichi Sakai
Graduate School of Information Science and Technology, University of Tokyo
Hongo 7-3-1, Bunkyo, Tokyo, Japan
{hung, sakai}@mtl.t.u-tokyo.ac.jp

Abstract

The amount of Task Level Parallelism (TLP) in runtime workload is useful information to determine the efficient usage of multiprocessors. This paper presents mechanisms to dynamically estimate the amount of TLP in runtime workloads. Modifications are added to the operating system (OS) to collect information about processor utilization, task activities, from which TLP can be calculated. By effectively utilizing the Time Stamp Counter (TSC) hardware, the task activities can be monitored at fine time resolution, resulting in capability of estimation of TLP at fine granularity.

We implement the mechanisms on a recent version of Linux OS. Evaluation results indicate that the mechanisms can estimate TLP accurately for various kinds of workloads with small overheads.

1. Introduction

Multiprocessors are increasingly becoming prevalent. Multiprocessors emphasize on thread-level, or 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 execution workload. Multiprocessors are beneficial when the amount of TLP in the workload is large. By executing the comprising 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 workload with poor TLP. Since there are various paradigms of thread usage other than to exploit parallelism [1], multithreaded applications with poor TLP are not uncommon [2][3]. Simply allocating many processors for executing such applications does not improve performance, but lead to power inefficiency. Some processors would often stay idle having their tasks waiting for data or synchronization from the counterpart tasks executing on other processors. The idle processors require portions of power consumption.

There are power consumed in maintaining cache coherency [4], or power consumed due to leakage. The leakage power already accounts for more than 30% of total power consumption in 90nm technology [5], and increases rapidly with technology scaling [6]. When the TLP is poor, activating only a few processors for execution and shutting down the other processors definitely offer lower power consumption with insignificant performance degradation.

The amount of TLP in a workload is therefore important information to determine 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 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 [3][8]. Hooks are added to the operating system (OS) [3], or multithreaded libraries [8] to collect information about tasks, or processor utilization. TLP is then calculated offline 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 these work. However, when TLP is dynamically estimated for any online optimization, the factors become very important. It is unclear that these existing mechanisms can be applicable for dynamic estimation of TLP.

This paper presents mechanisms to dynamically estimate 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 such information. By utilizing the Time Stamp Counter (TSC) hardware, the process activities can be easily monitored in detail, resulting in estimation of TLP at fine granularity with low overhead. Since no modification is required to applications and libraries, implementations of the mechanisms are transparent.
We implement the mechanisms on Linux OS. Evaluation results indicate that our technique can estimate TLP accurately for various workloads. The overheads imposed by the mechanisms are very small.

Section 2 describes the estimation targets, and the definitions of two kinds of TLP. Section 3 presents the estimation mechanisms. Implementation of the mechanisms in Linux OS 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 work. Since estimation of TLP is basically based on monitoring activities of tasks, we briefly cover task states and state transitions. We then describe the definitions of inter-processor TLP and intra-processor TLP, which are 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 done by OS based on the estimated parallelism. While the mechanisms does not estimate 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 cause much limitation. Since no modification is required for applications and libraries, implementation of the mechanisms is transparent.

2.2. Task States and State Transitions

Typical states that a task can have are: running, ready, and blocked. When a task runs out of its timeslice, or is preempted by a high priority task, the OS temporarily stops the running task and retreats it to a runqueue (running→ready transition). The OS then selects a ready process from the runqueue and lets the selected one run (ready→running transition). When a running task needs to wait for an event, it will be pending on a waitqueue (running→blocked transition). The processor may now turn 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 that 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 previous paper[3].

Intra-processor TLP, or intra-TLP for short, is defined as the TLP of those tasks that are executed on a 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. Alternately, 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 to estimate inter-TLP and intra-TLP are described in the following section.

3. Inter-TLP and Intra-TLP Estimation Mechanisms

Inter-TLP and intra-TLP also requires different mechanisms to estimate. 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 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 using Equation 1.

\[
TLP = \frac{\sum_{i=1}^{n} c_i i}{\sum_{i=1}^{n} c_i}
\]

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 1 shows an example of how inter-TLP can be calculated. Figure 1a shows the utilization of a multiprocessors system consisting of three processors P1, P2, and P3.
The dark, or white durations respectively indicate durations the processor is busy, or idle. In Figure 1.b, the utilization of each processor is quantized in time slot granularity. How many processors are busy in each slot are shown in Figure 1.c. In this example, the numbers of time slots that exactly one, two, and three processors are busy are respectively 5, 6, and 2. Following the Equation 1, the inter-TLP for this example is \( \frac{5 + 6 + 2}{5 + 6 + 2} = 1.77 \).

It is clear from Figure 1 that the accuracy of inter-TLP estimation depends on the size of the time slot. Using fine time slot improves the accuracy at the expense of increased estimation overhead. Such 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 executing on a 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 the shortage of processors, but is not due to any true data dependency among tasks. Removing ready durations from a real trace creates a new trace consisting of only blocked and running durations. The new trace is called imaginary trace. Whereas there is no overlap among running durations in the real traces of tasks executed on a same processor, the running durations in their imaginary traces can overlap. The resulting overlap represents the intra-TLP.

After the imaginary traces are generated, the degree of overlap among running durations of the imaginary traces is derived. The intra-TLP is then calculated also based on Equation 1. The \( c_i \) in this case represents the number of slots having \( i \) tasks in the running state indicated in the imaginary traces.

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

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

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

### 4. Implementation

We implement the proposed mechanisms on Linux OS. To the Linux OS, there is no concept of kernel-level thread; Linux implements all threads as normal processes [9]. 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 Equation 1. The resulting inter-TLP is output and the array is reset at the end of the period to be ready for next observation period.

**4.2. Implementation of Intra-TLP Estimation**

In Linux OS, many process-related information are kept in instances of task struct, one instance per process. Variables blocked_ts, ready_ts, and img_trace_len are newly added into task struct. The timestamps that a process enters, or exits the blocked state are respectively recorded in blocked_ts and ready_ts. img_trace_len represents the length of the process’s imaginary trace. The length of the blocked duration, which is equivalent to ready_tsblocked_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. 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 one. The value of img_trace_len is then increased by one. Updates of the array and img_trace_len is incrementally done by an interrupt timer and also by the context switch routine.

The number of elements of the bit array having their bits set, collected at the end of the observation period, is the value of the denominator in Equation 1. The numerator in Equation 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 img_trace_len of each process are reset accordingly.

**4.3. Time Keeping using Time Stamp Counter**

The size of the time slot affects the accuracy of the estimated TLP. Small time slot enables estimation of TLP at fine granularity. However, if existing time management facility in the OS (e.g., the gettimeofday system call) is used for high-resolution time management in our estimation mechanisms, frequent timing requests unavoidably incur large overhead.

Our TLP estimation mechanisms instead manages timing by utilizing the Time Stamp Counter (TSC). TSC is equipped in most of recent processors. TSC is a counter having its value increased by one in every clock cycle. Access to TSC is simply a matter of reading an on-chip register. Desired timing resolution is easily obtained by reading the TSC and shifting the result appropriately. For instance, for
Table 1. Evaluation Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-threaded benchmarks</td>
<td></td>
</tr>
<tr>
<td>mcf, twolf, bzip2</td>
<td>integer applications in SPECCPU 2000</td>
</tr>
<tr>
<td>Multithreaded benchmarks</td>
<td></td>
</tr>
<tr>
<td>mencoder</td>
<td>video encoder software</td>
</tr>
<tr>
<td>mozilla</td>
<td>web browser</td>
</tr>
<tr>
<td>dvdplayer</td>
<td>a DVD playback</td>
</tr>
<tr>
<td>openssl</td>
<td>cryptography toolkits</td>
</tr>
<tr>
<td>pigimp</td>
<td>benchmark for GIMP</td>
</tr>
</tbody>
</table>

1 GHz processor, reading the TSC and shifting the result to right by 15 bits achieve time resolution in 32 $\mu$s. The overhead of maintaining time in this manner is very small.

5. Evaluation

5.1. Evaluation Methodology

Our evaluation machine is a two-way multiprocessors 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 workloads consists of independent tasks from different single-threaded applications. The second type of workloads are multithreaded applications. Table 1 shows the list of the benchmarks including their 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 intra-TLP estimation mechanism can be evaluated on an uniprocessor without losing generality. The impacts of varying the size of the time slot on 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 those of the same workloads executed on the original OS. Applications from SPECINT benchmarks are chosen for overhead evaluations.

5.2. Results

5.2.1. Inter-TLP Estimation Results

Figure 3 shows the estimated inter-TLP for several workloads. The size of time slot is varied among 1000 $\mu$s, 125 $\mu$s, and 31 $\mu$s.

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 mcf+twolf workload almost equal to 2, is reasonable. However, since number of applications in the mcf+twolf+bzip2 workload is larger than 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 [1]. mozilla exhibits a bit higher TLP than other multithreaded applications.

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

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

5.2.2. Intra-TLP Estimation Results

Figure 4 shows the estimated intra-TLP. The size of the time slot is also varied among 1000 $\mu$s, 125 $\mu$s, and 31 $\mu$s. For reference, the inter-TLP estimated with time slot of 31 $\mu$s, derived from the results in Section 5.2.1, is also shown in Figure 4.

For 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. Note-
worthy, while the inter-TLP for mcf+twolf+bzip2 is estimated as 2, the intra-TLP on the other hand is estimated as 3, which correctly matches with the real TLP of the workload. Since the TLP extracted from concurrently execution of independent tasks represents the most common type of TLP found in practice, the capability of the mechanism to correctly estimate the TLP is encouraging.

For multithreaded applications, the estimated intra-TLP closely matches with the inter-TLP. Since those multithreaded applications exhibits poor TLP, the estimated inter-TLP would represent the real TLP of these applications. The estimation of intra-TLP is therefore accurate. Smaller time slot improves the accuracy of the estimation, as indicated by pigimp.

Table 3 shows the overhead of inter-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 fairly small (e.g., as much as 0.35%).

6. Conclusions

This paper presents two mechanisms for dynamically estimating the amount of Task Level Parallelism (TLP) in runtime workloads. The first estimates TLP among tasks executed on different processors of a multiprocessor system, and the second estimates TLP among tasks executed on a same processor. Appropriate modifications are added to the OS to collect information about processor utilization, task activities, based on which the TLP is deduced. Utilization of Time Stamp Counter (TSC) allows such information to be collected at high time resolution with low overhead.

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

Acknowledgement

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

References