



Intel[®] Technology Journal

Multi-Core Software

Process Scheduling Challenges in the Era of Multi-Core Processors

Process Scheduling Challenges in the Era of Multi-core Processors

Suresh Siddha, Software Solutions Group, Intel Corporation
Venkatesh Pallipadi, Software Solutions Group, Intel Corporation
Asit Mallick, Software Solutions Group, Intel Corporation

Index words: multi-core, chip multi-processors, process scheduling, power management

ABSTRACT

In this era of computing, each processor package has multiple execution cores. Each of these execution cores is perceived as a discrete logical processor by the software. Any operating system that is optimized for Symmetric Multi Processing (SMP) and that scales well with the increase in processor count can instantaneously benefit from these multiple execution cores.

Design innovations in multi-core processor architectures bring new optimization opportunities and challenges for the system software. Addressing these challenges will further enhance system performance. The process (task) scheduler, in particular, one of the critical components of system software, is garnering great interest.

In this paper, we look at how the different multi-core topologies and the associated processor power management technologies bring new optimization opportunities to the process scheduler. We look into different scheduling mechanisms and the associated tradeoffs. Using the Linux* Operating System as an example, we also look into how some of these scheduling mechanisms are currently implemented.

As the multi-core platform is evolving, some portions of the hardware and software are being reshaped to take maximum advantage of the platform resources. We close this paper with a look at where future efforts in this technology are heading.

INTRODUCTION

In multi-core processor packages, each processor package contains two or more execution cores, with each core having its own resources (registers, execution units, some or all levels of caches, etc.). Even if the applications are not multi-threaded, multi-tasking environments will benefit from multi-core processors.

Design innovations of multi-core processor architectures mainly span the area of shared resources (caches, power management, etc.) between cores, core topologies (number of cores in a package, relationship between them, etc.), and platform topology (relation between cores in different packages, etc.). These innovations bring new opportunities and challenges to the system software. To exploit optimal performance, components such as the process scheduler need to be aware of the multi-core topologies and the task characteristics.

We start with a brief look at how the traditional process scheduler works and how the earlier challenges in the Symmetric Multi Processing (SMP), Non Uniform Memory Access (NUMA), and Simultaneous Multi-Threading (SMT) environments were addressed. We look at multi-core topologies with respect to core, cache, power management, and platform topologies. In the current generation of mainstream multi-core processors, the execution cores in a given processor package are symmetric and our focus in this paper is on such processors. Asymmetric multi-core processors are beyond the scope of this paper. We examine the need for a multi-core-aware process scheduler and look into the opportunities in this area. We examine different scheduling mechanisms for multi-core platforms under different load scenarios and the associated tradeoffs. With Linux as an example, we examine how some of these scheduling mechanisms are currently implemented. Finally, we close this paper with a look at current and future research in this field.

PROCESS SCHEDULER

The process scheduler, which is a critical piece of the operating system software, manages the CPU resource allocation to tasks. The process scheduler typically strives

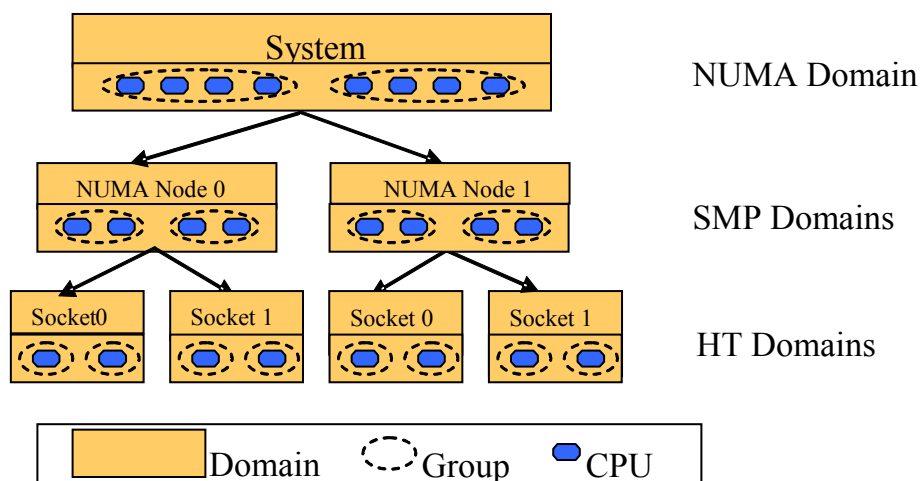


Figure 1: Process scheduling domain topology setup in the Linux kernel

for maximizing system throughput, minimizing response time, and ensuring fairness among the running tasks in the system.

Process priority determines the allotted time (time-slice) on a CPU and when to run on a CPU. In SMP, the process scheduler is also responsible for distributing the process load to different CPUs in the system.

In NUMA platforms, memory access time is not uniform across all the CPUs in the system and depends on the memory location relative to a processor. System software tries to minimize the access times, by allocating the process memory on the node that is closest to the CPU that the process is running on. As such, the cost associated with the process migration from one NUMA node to another is big. As a result, the process scheduler needs to be aware of NUMA topology. NUMA schedulers use some heuristics (such as tolerating more load imbalances between nodes and tracking the home node of each process, where the majority of process memory resides) to minimize the migrations and costs associated with the migrations.

In SMT (for example, Intel[®] Hyper Threading Technology), most of the core execution resources are shared by more than one logical processor. The process scheduler needs to be aware of the SMT topology and avoid situations where more than one thread sibling on one core is busy, while all the thread siblings on another core are completely idle. This will minimize the resource contention, maximize the utilization of CPU resources, and thus maximize system throughput. As the logical thread siblings are very close to each other, process migration between them is very cheap and as such, process load balancing between them can be done very often.

The process scheduler needs to consider all these topological differences while balancing process loads across different CPUs in the system. For example, the 2.6 Linux kernel process scheduler introduced a concept called scheduling domains [8] to incorporate the platform topology information into the process scheduler. The hierarchical scheduler domains are constructed dynamically depending on the CPU platform topology in the system. Each scheduler domain contains a list of scheduler groups having a common property. The load balancer runs at each domain level, and domain properties dictate the balancing that happens between the scheduling groups in that domain. On a high-end NUMA system with SMT capable processors, there are three scheduling domains, one each for SMT, SMP, and NUMA, as shown in Figure 1.

MULTI-CORE TOPOLOGIES

In most of the multi-core implementations, to make the best use of the resources and to make inter-core communication efficient, cores in a physical package share some of the resources. For example, the Intel[®] Core[™]2 Duo processor has two CPU cores sharing the Level 2 (L2) cache (Intel[®] Advanced Smart Cache), as shown in Figure 2. The Intel[®] Core[™]2 Quad processor has four cores in a physical package with two last-level (L2) caches. Each of the L2 caches is shared by two cores. Going forward, as more and more logic gets integrated into the processor package; more resources will be shared between the cores on the die.

If only one of the cores in the package is active, a thread running on that core gets to use all the shared resources, resulting in maximum resource utilization and peak performance for that single thread. If multiple threads or

processes run on different cores of the same physical package and if they share data that fit in the cache, then the shared last-level cache between cores will minimize the data duplication. This sharing, therefore, results in more efficient inter-thread communication.

Multi-core Power Management

In typical multi-core configurations, all cores in one physical package reside in the same power domain (voltage and frequency). As a result, the processor performance state (P-state) transitions for all the cores need to happen at the same time. If one core is busy running a task at P0, this coordination will ensure that other cores in that package can't enter low-power P-states, resulting in the complete package at the highest power P0 state for optimal performance.

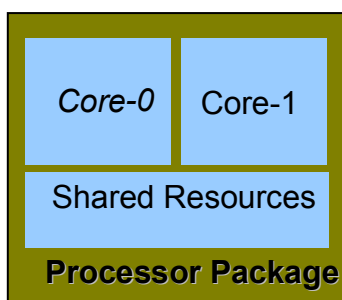


Figure 2: Dual-core package with shared resources

Since each execution core operates independently, each core block can independently enter a processor power state (C-state). For example, one core can enter lower power C1 or C2 while the other executes code in the active power state C0. The common block will always reside in the numerically lowest (highest power) C-state of all the cores. For example, if one core is in C2 and another core is in C0, the shared block will reside in C0.

Intel Dynamic Acceleration Technology

Intel[®] Dynamic Acceleration Technology [7], available in the current Intel Core 2 processor family, increases the performance of single-threaded applications. If one core is in deep C-state, some of the power normally available to that idle core can be applied to the active core while still staying within the thermal design power specification for the processor. This increases the speed at which a single-threaded application can be executed, thereby improving the performance of the application.

MULTI-CORE SCHEDULING

Shared resource topologies in multi-core platforms pose interesting challenges and opportunities to the system software. Shared resources between cores like shared cache hierarchy, provide good resource utilization and

make inter-core communication efficient. However, heterogeneous data access patterns of memory-intensive tasks running on the cores sharing caches can lead to cache contention and sub-optimal performance. Contention and its impact on performance depend on the resources shared, the number of active tasks, and the access patterns of the individual tasks. A fair amount of CPU time allocated to each task by the process scheduler will not essentially translate into efficient and fair usage of the shared resources. The main challenge before the process scheduler is to identify and predict the resource needs of each task and schedule them in a fashion that will minimize shared resource contention, maximize shared resource utilization, and exploit the advantage of shared resources between cores. To achieve this, the process scheduler needs to be aware of multi-core, shared resource topology, resource requirements of tasks, and the inter-relationships between the tasks.

In the following sections, we describe some of the multi-core scheduling mechanisms; challenges in exploiting optimal performance, and power savings in the SMP platform. We analyze the impact of Intel Dynamic Acceleration Technology and processor power management technologies on these scheduling mechanisms. We also look into some of the heuristics that today's system software can exploit to minimize the shared resource contention among the cores sharing resources.

Experimental Setup

For our experiments and analysis, we primarily considered a dual-package SMP platform, with each package having two cores sharing a 4MB last-level cache. Different workloads such as SPECjbb2000, SPECjbb2005, SPECfp_rate of CPU2000, and an in-memory database search (IMDS) are considered for our analysis. These workloads are widely known except for the IMDS one. The IMDS workload is a non-standard workload simulating CPU and memory behavior of a typical database search algorithm. This workload is considered because of its high cycles per instruction (CPI) characteristic when compared to the other workloads used in our experiments. Run to run variations of these workloads are within 1%. Each of these workloads was run three times and the middle number was used for the performance comparisons.

Some of these workloads (SPECjbb2000, IMDS, for example) spawn threads sharing a process address space and some (like SPECfp_rate) spawn different processes, each having its own address space. Platform under test is run at 3GHz processor frequency unless otherwise stated and doesn't support Intel Dynamic Acceleration Technology.

Scheduling on Cores Sharing Resources vs. Not Sharing

In this workload scenario, all the considered workloads were run in a 2-task configuration. For example, the SPECjbb workload was run in two warehouse configurations (where each warehouse is represented by a user-level thread), and the SPECfp rate was run in the base configuration with two users (where each user is represented by an individual process). Similarly, the IMDS workload used two threads (belonging to the same process) to process the database queries.

Table 1: Performance difference between scheduling two tasks running on two cores using different last-level cache vs. scheduling on cores sharing same last-level cache. The higher % means that scheduling on cores with different caches is better.

Workloads	% Performance improvement with scheduling on different last-level caches when compared to scheduling on same last-level caches
SPECjbb2005	13.22
SPECjbb2000	5.19
SPECfp_rate (base2000)	16
IMDS	1

With two running threads on a dual-core, dual-package SMP platform, the main choices before the process scheduler are to schedule the two running threads on the cores in the different (Option 1) or same (Option 2) packages. Option 1 will result in maximum resource utilization, and as the other core in each package is idle, there is no shared resource contention. Option 2 will result in one busy package (with both the cores busy running the tasks), and the other package being completely idle. While this is not the best solution from the resource utilization and shared resource contention (tasks running in one package may contend for shared resources between cores) perspective, this mechanism will take advantage of the data sharing between tasks, if any.

Table 1 shows the results of different workloads with different scheduling mechanisms on a dual-core, dual-package SMP platform. As shown in the table, all the workloads benefited from distributing the load to two different packages. This indicates that the considered workloads take advantage of the increased available cache and the shared resource (primarily last-level cache) contention is playing a significant role when both the tasks run on the same package. Moreover, the contention is present whether the running tasks belong to the same process (where there is some data sharing, for example,

SPECjbb) or different processes (for example, SPECfp_rate of cpu2000). Among the workloads, the IMDS workload in fact performs almost the same, irrespective of the threads running on cores sharing the same or different packages. This is primarily because the workload doesn't exhibit good locality of memory references and as such doesn't get affected much by sharing the last-level cache between two threads.

Last-level Cache Size Influence on Shared Resource Contentions

Hardware designers and researchers are looking into different options (like optimum size, layout of shared resources, design and management of these shared resources) and solutions for maximizing resource utilization and at the same time minimizing the resource contention. Moore's law [6] is dictating the cache size increase on Intel® platforms from generation to generation. The current x86 generation of 65nm processors features up to 4MB of L2 cache in the dual-core version and up to 8MB in the quad-core version, and the leading-edge 45nm generation [1] of x86 processors sports up to 6MB of L2 cache in the dual-core version and up to 12MB in the quad-core version. The degree of cache associativity is increasing with the increase in cache size, leading to hit rate improvement and better utilization.

Figure 3 shows the impact of the last-level cache size on the process scheduling mechanisms for the workloads we considered in Table 1. While the platforms considered for this experiment are quite different from each other (different characteristics and properties), each of the platforms under test is configured to work as dual-core, dual-package platforms. Each of these platforms has a different last-level cache size shared by two cores that reside in the processor package.

The data in Figure 3 show that for the given task load (two tasks in our experiments), as the shared resources among the cores increases, one can expect that the amount of shared resource contention will decrease accordingly. For example, SPECfp_rate of CPU2000 was performing the same whether the two tasks were running in the same package or in different packages with 16MB of last-level cache. The impact of last-level cache size is fairly negligible for the two threaded IMDS workloads. As noticed before, this is primarily because of the poor locality of memory references.

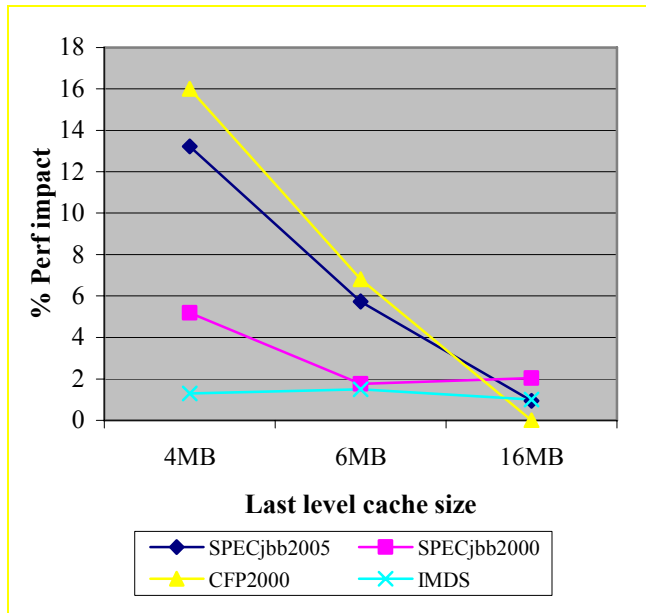


Figure 3: Impact of last-level cache size on the performance differences between scenarios doing scheduling on cores using same vs. different last-level cache. The smaller number indicates less resource contention in the scenarios when tasks run on cores sharing last-level cache.

Influence of Intel Dynamic Acceleration Technology

Intel Dynamic Acceleration Technology is currently available in Intel Core 2 Duo mobile processors. Let us look into the influence of this technology on process scheduling, if this support is available in the future for the server platforms supporting multiple processor packages.

Using the Linux kernel CPUfreq subsystem, we simulated the concept of Intel Dynamic Acceleration Technology in today’s mainstream dual-package platform based on Intel Core 2 Duo processors. With the help of CPUfreq subsystem, the processor frequency can be changed to a specific value that the processor supports. Using this infrastructure, 3GHz-capable processors were run at 2.66GHz (a bin down) in the mode when the process scheduler schedules the two running tasks on two cores belonging to the same package. In the mode when the scheduler selects two different packages for running the two tasks, processors were run at 3GHz (as Intel Dynamic Acceleration Technology will enhance the speed of the active core, while one or more cores in the same package are idle). Figure 4 shows the performance numbers, which include the effects of running on different caches and at improved processor speeds as a result of dynamic acceleration.

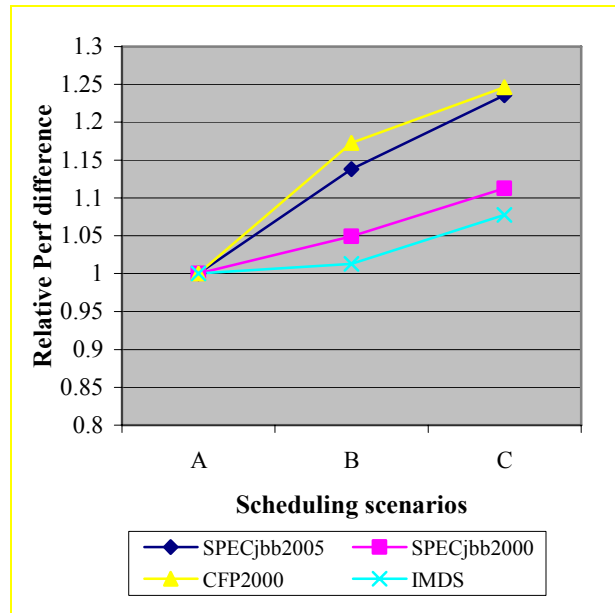


Figure 4: Performance difference between running two tasks on a) cores running at 2.66GHz, sharing last-level cache vs. b) cores running at 2.66GHz, different last-level cache vs. c) cores running at 3GHz, different last-level cache

Figure 4 shows that the dynamic acceleration favors the scheduling policy of distributing the load among the available processor packages for optimal performance. In the presence of dynamic acceleration, IMDS workloads also benefited when the two IMDS threads were run on different packages.

Scheduling for Optimal Power Savings

Consider the same dual-package experimental system that we looked at before. If we have two runnable tasks, as observed in the previous sub sections, resource contention will be minimized when these two tasks are scheduled on different packages. But, because of the P-state coordination in the current generation of multi-core platforms, we are restricting other idle cores in both packages to run at higher power P-state (voltage/frequency pair). Similarly, the shared block in both packages will reside in higher power C0 state (because of one busy core). This will result in a non-optimal solution from a power-savings perspective. Instead, if the scheduler picks the same package for both tasks, other packages with all cores being idle, will transition into the lowest-power P and C-state, resulting in more power savings. For optimal power savings, the number of physical packages carrying the load needs to be minimized. But as the cores share resources (like last-level cache) as seen in previous sections, scheduling both tasks

to the same package may or may not lead to optimal behavior from a performance perspective.

Task Group Scheduling

In the workload scenario where all the shared resources and packages are busy, the main challenge for the process scheduler is to schedule the tasks in such a way that will minimize the shared resource contention and take maximum advantage of the shared resources between cores.

If all the running tasks are resource intensive, the challenge before the process scheduler is to identify the tasks that share data and schedule them on the cores sharing the last-level cache. This will help minimize the shared resource contentions and shared data duplication. This will also result in efficient data communication between the tasks that share data. The system software has some inherent knowledge about data sharing between tasks. For example, threads belonging to a process share the same address space and as such share everything (text, data, heap, etc.). Similarly, processes attached to the same shared memory segment will share the data in that segment. The process scheduler can do optimizations such as grouping threads belonging to a process or grouping processes attached to the same shared memory segment and co-schedule them in the cores sharing the package resources. To highlight the group-scheduling potential, we ran two instances of the SPECjbb workload, with each instance having two warehouses (each warehouse represented by a thread) on the dual-core, dual-package platform, considered before. Table 2 shows that grouping the threads belonging to a process onto the same package takes advantage of shared resources between cores and helps minimize the shared resource contentions.

Table 2: Performance improvement seen when threads belonging to a process scheduled to two cores residing on same package when compared to scheduling them on different packages. Workload considered is with two instances of SPECjbb in a two warehouse (two processes with two threads each) configuration.

Workloads	% Throughput improvement
SPECjbb2005	10
SPECjbb2000	7.5

Scheduling Challenges

For exploiting optimal performance, the process scheduler needs to schedule tasks in such a way that all the platform resources are used effectively. And this effective mechanism varies with workload, processor, platform topology, and system load.

Some workloads will exploit optimal benefit when the tasks are scheduled on the cores that share resources. For example, the IMDS workload performed the same, whether the tasks were run on cores sharing resources or not. For such workloads, in the presence of idle packages, by scheduling the tasks on the cores residing in a package, optimal performance and power-savings will be achieved. Similarly, workloads that share data and take maximum advantage of the shared resources between cores will achieve optimal performance when run on cores that are closer. For example, if the data shared between the tasks are modified and exchanged often or if one executing task prefetches data for the other task, optimal performance will be achieved when the tasks are scheduled closer to each other.

For some workloads, even in the presence of data sharing, distributing the load among the available idle packages will lead to optimal performance. This distribution will lead to shared data duplication in the caches of the packages carrying the load. If the shared data are mostly read-only, this data duplication still may be better than leaving the shared resources idle. In this scenario, tasks can take advantage of the increased shared resources (caches in our considered setup) that are available and can cache more shared data and/or task private data.

The presence of technologies, like dynamic acceleration, influence the process scheduling mechanisms for some workloads in the presence of idle cores and packages. As seen in Figure 4, when the load is uniformly distributed among the available packages, the resulting core speed increases, resulting from dynamic acceleration, helped achieve optimal performance. For some workloads, even in the presence of dynamic acceleration, running on the cores sharing caches may give optimal performance.

In future, as more cores are integrated into the processor package, the available shared resources will also increase accordingly. As such, the amount of shared resource contention will be minimal when few of the available cores in the package are busy (similar to what we see in Figure 3). The challenge for the process scheduler is to track the shared resource usages and the associated contentions. In such a scenario, the scheduler can minimize the processor packages carrying load, and when there is a contention for the shared resources, the scheduler can distribute the load to minimize resource contentions. To address the challenge, the process scheduler needs to track the micro-architectural information like the task's cycles per instruction (CPI) and how the task's CPI is affected by the co-running tasks in the other cores sharing resources. An individual task's CPI will also help the process scheduler in making decisions such as which tasks benefit most from the increased core speed that the dynamic acceleration technology brings in.

In a scenario where all the shared resources and packages are busy, the process scheduler needs to minimize the resource contention for exploiting optimal performance. For example, grouping CPU-intensive and memory-intensive tasks onto the cores sharing the same last-level cache will result in minimized cache contention. Task characteristics and behavior can be predicted using the micro-architectural history of a task by using performance counters. In the absence of such micro-architectural information, the system software can also use some heuristics to estimate the resource requirements. For example, one can use the number of physical pages that are accessed (using the Accessed bit in the page tables that manage virtual to physical address translation in x86 architecture) for certain intervals or use the tasks memory Resident Set Size (RSS). The process scheduler can use this information and group schedule tasks on the cores residing in a physical package with the goal of minimizing shared resource contention.

MULTI-CORE AWARE LINUX* PROCESS SCHEDULER

In this section, we consider the Linux operating system as an example and see how some of these scheduling challenges are addressed. A new scheduler domain representing multi-core characteristics has been added to the domain hierarchy of the Linux process scheduler. This scheduler domain helps identify cores sharing the same package and sharing resources, and it paves the way for the multi-core scheduler enhancements.

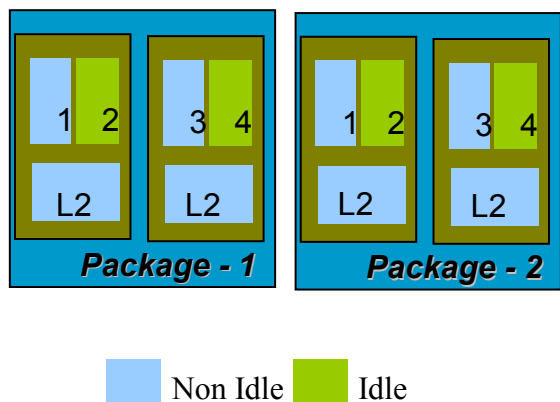


Figure 5: Scheduling mechanism showing four running tasks scheduled on four L2s on a dual package with Intel Core 2 Quad processors

By default, the current Linux kernel scheduler distributes the running tasks equally among the available last-level caches in an SMP domain. Within logical CPUs that share the last-level cache, the scheduler distributes the load equally, first among the available CPU cores and then

among the available logical thread siblings. For example, consider a dual package SMP platform with Intel Core 2 quad processors with four running tasks. The multi-core-aware Linux process scheduler distributes these four running tasks among the four L2's that are available in the system as shown in Figure 5. This scheduling mechanism will lead to maximized resource utilization and minimized resource contention. And as observed in the previous sections, this will lead to optimal performance for most of the workloads. On platforms with dynamic acceleration technology, this mechanism will also result in optimal performance by making the cores run faster.

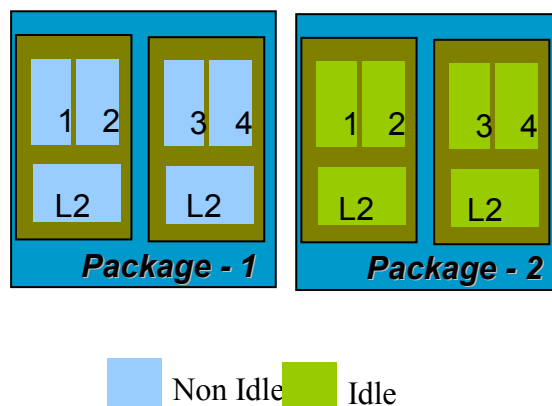


Figure 6: Scheduling mechanism showing four running tasks scheduled on four cores in one single package on a dual package with Intel Core 2 Quad processors

For optimal power savings or for workloads that benefit most by running the tasks on the cores sharing resources, the Linux kernel provides a tunable that can be set by an administrator. When this tunable is set, the process scheduler will try to minimize the packages in an SMP domain that carry the load. It will first try to load all the logical threads and cores in the package before distributing the load to another package. This policy is referred to as a power-savings policy. For example, consider the same four-task scenario on a dual package SMP platform with Intel Core 2 Quad processors. With the power-savings tunable set, all the four tasks will be run on the four cores residing in a single package as shown in Figure 6. Minimizing the number of packages that are active will lead to optimal power savings. As seen before, in the absence of dynamic acceleration support, this scheduling mechanism will not have any impact on performance for workloads such as the IMDS workload considered in Table 1.

Scheduling policy decisions are left to the administrator in the hope that the target workloads will be analyzed offline and the tunable will be set based on optimal performance and/or power-savings requirements. By default, the

process scheduler takes a non-aggressive approach when distributing the load among the available shared resources.

RESEARCH WORK

Quite a bit of recent research in the process scheduler area is to do with trying to address multi-core scheduling challenges. For example, Micro Architectural Scheduling Assist [2] talks about tracking the shared resource usage with performance-monitoring counters and using this information for effective distribution of shared resource load. Another body of work in this area is the cache-fair algorithm [4] that tries to address the application performance variability that depends on the other co-scheduled threads in the same multi-core package. This algorithm uses an analytical model to estimate the L2 cache miss rate a thread would have if the cache were shared equally among all the threads, i.e., the fair miss rate. The algorithm then adjusts the thread's share of CPU cycles in proportion to its deviation from its fair miss rate. This algorithm showed a reduction of the effect of the schedule-dependent miss rate variability on the thread's runtime. The L2-conscious scheduling algorithm [5] separates all runnable threads into groups, such that the combined working set of each group fits in the cache. By scheduling a group at a time and making sure that the working set of each scheduled group fits in the cache, this algorithm reduces the cache miss ratios.

While the research shows promising results, it is far from being implementation ready and from inclusion in commercial operating systems. The main challenges of these algorithms include the dependency of the performance-monitoring counters (which are not designed primarily for process scheduling and which vary from processor generation to processor generation), the different algorithm phases (data collection phase and usage phase), applicability of mathematical models to wide heterogeneous workloads, and above all, incorporating this knowledge into the traditional process scheduler that works across wide multi-core topologies and platforms. One of the current focus areas is to turn this research into reality.

Most of the software algorithms exploit the differences in the individual task characteristics and their resource usages. Scenarios such as those in which all the tasks in the system have similar characteristics and resource requirements cannot be addressed by software alone with the current generation of multi-core hardware. CQoS [3] presents a new cache management framework for improving shared cache efficiency and improving system performance. It proposes options for priority classification, priority assignment, and priority enforcement to heterogeneous memory access streams. Hardware solutions like these help maximize resource

utilization and minimize the impact on performance in the presence of shared resource contention.

As more logic gets integrated into the processor die, future work in this area will focus on the increasing shared resources between cores on the die and their interactions with the system software; the process scheduler in particular. In the area of multi-core processor power management, one of the areas that is making rapid progress is the reduction of idle processor power. In future platforms, as the power consumed by idle cores decreases and becomes independent of the busy cores in the packages, scheduling mechanisms for power savings need to be revisited.

CONCLUSION

In this paper, we showed that optimal performance can be exploited by making the process scheduler aware of the multi-core topologies and the task characteristics. Multi-core scheduling mechanisms and challenges are analyzed in an SMP environment. We looked at the impact of Intel Dynamic Acceleration Technology on these workload scenarios. Some of the group-scheduling heuristics that can enhance optimal performance are presented. We looked at how some of these multi-core scheduling mechanisms are implemented in the Linux operating system.

In future, one can expect the process scheduler to be micro-architectural aware for exploiting optimal performance. Similarly, one can expect that the research proposals and solutions in this area will drive future hardware and platform designs that will minimize the effects of shared resource contention and also assist the software in making and enforcing the right decisions.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their feedback. We thank David Levinthal for his mentoring support.

REFERENCES

- [1] "Introducing the 45nm Next-Generation Intel® Core™ Microarchitecture," at http://www.intel.com/technology/architecture-silicon/intel64/45nm-core2_whitepaper.pdf
- [2] Nakajima, Jun and Pallipadi, Venkatesh, "Enhancements for Hyper-Threading Technology in the Operating System—Seeking the Optimal Scheduling," *Workshop on Industrial Experiences with Systems Software*, Boston, MA, Dec. 2002.
- [3] R. Iyer, "CQoS: A Framework for Enabling QoS in Shared Caches of CMP Platforms," *18th International*

Conference on Supercomputing (ICS), San Malo, France, June 2004.

- [4] Alexandra Fedorova, Margo Seltzer, Christopher Small and Daniel Nussbaum, "Performance Of Multithreaded Chip Multiprocessors and Implications For OS Design," in *Proceedings of the USENIX 2005 Annual Technical Conference*, Anaheim, CA, April 2005.
- [5] Alexandra Fedorova, "Improving Performance Isolation on Chip Multiprocessors via an OS Scheduler," at <http://www.eecs.harvard.edu/~fedorova/papers/fairsched.pdf>
- [6] "Moore's Law," at <http://www.intel.com/technology/mooreslaw/index.htm>
- [7] "Intel® Centrino® Duo Processor Technology," at <http://www.intel.com/products/centrino/duo/description.htm>
- [8] "Scheduling Domains," at <http://lwn.net/Articles/80911/>

AUTHORS' BIOGRAPHIES

Suresh Siddha is a Senior Staff Software Engineer in Intel's Open Source Technology Center. Suresh joined Intel in 2001 and works on enabling various Intel processor and platform features in the Linux kernel. His current focus is on multi-core technologies, the process scheduler, and system software scalability. His e-mail address is suresh.b.siddha@intel.com.

Venkatesh Pallipadi is a Senior Staff Software Engineer in Intel's Open Source Technology Center. He joined Intel in 2001 and works on enabling various core features in the Linux kernel across all Intel architectures. His current focus is on processor and platform power management. Prior to joining Intel, Venkatesh received his ME degree in Computer Science and Engineering from the Indian Institute of Science in Bangalore, India. His e-mail is venkatesh.pallipadi@intel.com.

Asit Mallick is a Senior Principal Engineer leading the system software architecture in the Intel Open Source Technology Center. He joined Intel in 1992 and has worked on the development and porting of numerous operating systems to Intel architecture. Prior to joining Intel, he worked in Wipro Infotech, India on the development of networking software. Asit earned his Masters degree in Engineering from the Indian Institute of Science, India. His e-mail address is asit.k.mallick@intel.com.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino logo, Core Inside, FlashFile, i960, InstantIP, Intel, Intel logo, Intel386, Intel486, Intel740, IntelDX2, IntelDX4, IntelSX2, Intel Core, Intel Inside, Intel Inside logo, Intel Leap ahead., Intel Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, IPLink, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skool, Sound Mark, The Journey Inside, VTune, Xeon, and Xeon Inside are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Intel's trademarks may be used publicly with permission only from Intel. Fair use of Intel's trademarks in advertising and promotion of Intel products requires proper acknowledgement.

*Other names and brands may be claimed as the property of others.

Microsoft, Windows, and the Windows logo are trademarks, or registered trademarks of Microsoft Corporation in the United States and/or other countries.

Bluetooth is a trademark owned by its proprietor and used by Intel Corporation under license.

Intel Corporation uses the Palm OS® Ready mark under license from Palm, Inc.

Copyright © 2007 Intel Corporation. All rights reserved.

This publication was downloaded from <http://www.intel.com>.

Additional legal notices at: <http://www.intel.com/sites/corporate/tradmarx.htm>.

THIS PAGE INTENTIONALLY LEFT BLANK

For further information visit:

developer.intel.com/technology/itj/index.htm