acmqueue

Extending the Semantics of Scheduling Priorities

Increasing parallelism demands new paradigms.

Rafael Vanoni Polanczyk, Oracle Solaris Kernel Group

Application performance is directly affected by the hardware resources that the application requires, the degree to which such resources are available, and how the operating system addresses its requirements with regard to the other processes in the system. Ideally, an application would have access to all the resources it could use and be allowed to complete its work without competing with any other activity in the system. In a world of highly shared hardware resources and general-purpose, time-share-based operating systems, however, no guarantees can be made as to how well resourced an application will be.

What can be done to improve both the way in which applications are developed and how the underlying layers of the software stack operate, in order to gain better overall utilization of shared hardware resources? Extending some of the semantics of scheduling priorities to include priority over shared resources could allow the performance-critical components of applications to execute with less-contention for the resources they require.

DYNAMICALLY RESOURCED, HIGHLY SHARED HARDWARE COMPONENTS

The past decade has seen the emergence of the multicore processor and its subsequent rapid commoditization. Software developers, whether at the application level or in the broader design of systems, simply cannot ignore this dramatic change in the landscape. While not all problems require a parallel implementation as a solution,¹ this opportunity must be considered more seriously than ever before. Both academia and the industry have followed this trend by educating and advocating concurrent software development, while also seeking new techniques to exploit hardware parallelism. Unfortunately, many details about developing concurrent applications on modern processors have slipped through the cracks, to be found only in white papers, blogs, and similar watering holes of the performance community.

What developers are finding more often than not is that sizing parallel applications is not as straightforward as it once seemed. Until quite recently the one or two processors available within a single processor chip did not cause more contention for shared resources than perhaps two software threads fighting over shared cache space. Nowadays, not only are there several levels of sharing between logical CPUs (shared execution pipeline, floating-point units, different cache levels, etc.), but also these many more CPUs make that sharing a much more complicated problem.

If this growing multiprocessing scale and its associated microarchitectural complexity weren't enough, modern processors are also dynamically adapting their processing capacity based on current utilization in an attempt to provide applications with the resources they need. Intel's Turbo Boost feature increases the processor's operating frequency as fewer cores are active and thermal conditions allow. The SPARC T4 processor, in contrast, dynamically allocates core resources to its active hardware threads, incrementally benefiting the few active threads by having more inactive

ones. Both features are in essence enabling heterogeneous applications by improving single-threaded performance.

This new landscape poses new questions for you, as both a developer and a system administrator: How many threads should your workload create? What resources do they need? Are all threads equally important? How should you size shared data structures? What should you tell the operating system (or more generically, the underlying layer) about your application?

PROVISIONING THREADS IN MULTITHREADED APPLICATIONS

Although the simple classic recipe of one software thread for each logical CPU may still be valid in some cases, it can no longer be applied indiscriminately.² Parallel applications must know which portions of a program may require resources that are not widely available in the system. With that knowledge and some understanding of the possible deployment platforms, applications may be able to size themselves by matching their hardware requirements to what the underlying layer has to offer. Failure to do this properly leads to either contention over shared resources—as too many threads compete for them—or the underutilization of available resources.

For homogeneous multithreaded applications—those in which all threads perform similar tasks (and therefore have similar requirements)—you could simply partition the available resources into n slices according to how much of each resource a single thread will require. A scientific application that makes heavy use of floating point might create one thread per available FPU (floating-point unit) in the system (or two or three, if they can all take turns while executing their floating-point sections). For heterogeneous workloads, on the other hand, it may be advantageous to set aside more resources for specific threads. For example, in a producer/consumer architecture with a single producer and various consumers, giving the producer as many resources as it can take advantage of would likely be beneficial, and help keep the consumers as busy as possible. This dependency relationship between producer and consumers is the primary point of contention in the application, making the producer its most critical component.

You may also want to exploit the knowledge of sharing relationships to take advantage of the dynamic resourcing features in recent processors. In other words, you can manually create the conditions that allow these features to come into play. In this case the goal is not simply to prevent performance degradation by reducing the number of threads competing for some necessary component; you want the application to take advantage of all the performance you can get from the processor. In the previous producer/consumer example, the throughput of the application would likely increase if you placed the producer on a dedicated core, granting it exclusive access to all the hardware resources within that component.

CONSEQUENCES OF VIRTUALIZATION

Virtualization mechanisms are a confounding factor, as they often hide the details of the underlying architecture and the current utilization levels of its various components (such as obscuring direct access to the physical performance counters on the processor). This may prevent an application from determining the available physical resources, and therefore make it unable to size itself correctly by distributing its requirements across the available resources. It may also prevent the application from monitoring the consequences of its own behavior and adapting to changes in system utilization. Thankfully, these limitations can be circumvented if the application uses its own mechanisms to

evaluate performance and capacity. For example, the application can run a micro-benchmark during startup, and/or periodically as it runs, to evaluate how the system is performing according to its own metrics. It can then use that information to adapt to the current conditions.

In the end, even a correctly sized parallel application still relies on the operating system—or on the underlying runtime environment—for mechanisms to provision threads, as well as for appropriate scheduling decisions for thread placement. Unfortunately, operating systems have traditionally offered only very simple mechanisms to provision specific threads. For example, the use of processor sets in Solaris to provide an entire core (and its otherwise shared components) to certain threads in an application is a reasonably well-known tuning method used by field engineers and specialized customers. Process binding is also used when manually placing processes and threads to ensure a desired behavior. These mechanisms are too static, however. They require manual intervention and are usually too expensive for this purpose. A preferable solution would be to provision threads more accurately with the resources they require as they become runnable, without user intervention, leveraging developers' knowledge of their applications and reducing the amount of work (or interference) required from the operating system or the system administrator.

PRIORITY OVER SHARED RESOURCES

The current implementation and semantics of scheduling priorities date from a period when single-processor systems were the norm. Resources were very limited and had to be correctly divided among threads in the system by allowing them to run for determined periods of time according to their priority. The recent emergence of systems with a large number of processors has fundamentally changed the scenario. Given the large number of resources available, threads no longer compete just for processor time, but also for shared hardware resources.

This scheduling model fails to recognize the sharing aspects of today's processors, producing some performance anomalies that can be difficult to address. Consider, for example, a high-priority thread competing for a specific resource with a set of "hungry" lower-priority threads. In this case, it would be desirable to extend the implementation of priorities to include priority over shared resources. The operating system could then choose to move the lower-priority threads away from where the higher-priority one is running or to find a more appropriate place for it to execute with less-contended resources.

This extension presents a new method through which developers and system administrators can specify which components of an application should be more or less provisioned. It's a dynamic, unobtrusive mechanism that provides the necessary information for the operating system to provision threads more effectively, reducing contention over shared resources and taking advantage of the new hardware features discussed previously. Furthermore, the new behavior is likely to benefit users who already identify threads in their applications with different levels of importance (an important aspect of this work, for practical reasons).

Additionally, several other aspects of priorities play to our advantage. Since the proposed "spatial" semantics will determine how many resources will be assigned to threads, it is critical that this mechanism be restricted to users with the appropriate privileges—already a standard aspect of priorities in all Unix operating systems. Priorities can also be applied at different levels: at the process, thread, or function level, allowing optimizations at a very fine granularity.

LOAD BALANCING AND PRIORITIES

Load balancing is a classic concept in scheduling. Modulo implementation details, the basic idea is to equalize work across execution units in an attempt to have an even distribution of utilization across the system. This basic assumption is correct, but the traditional implementation of load balancing does not perform well in heterogeneous scenarios unless the scheduler is capable of identifying the different requirements of each thread and the importance of each thread within the application.

A few years ago the Solaris scheduler was extended to implement load balancing across shared hardware components in an effort to reduce resource contention. We had discovered that simply spreading the load across all logical CPUs was not enough: it was also necessary to load-balance across groups of processors that shared performance-relevant components. To implement this policy, Solaris established the Processor Group abstraction. It identifies and represents shared resources in a hierarchical fashion, with groups that represent the most-shared components (pipe to memory, for example) at the top and groups that represent the least-shared ones (such as execution pipeline) at the bottom. Figure 1 illustrates the processor group topology for two different processors: the SPARC T4 and Intel Xeon processors, with each hardware component and the CPUs they contain.

Each processor group maintains a measure of its capacity and utilization, defined as the number of CPUs and running threads in a group. This information is incorporated by the scheduler and used when deciding where to place a software thread, favoring groups where the utilization:capacity ratio would allow the thread to make the most progress.

PERFORMANCE-CRITICAL THREADS

The processor group abstraction and the associated load-balancing mechanism for multicore, multithreaded processors successfully reduced contention at each level of the topology by spreading the load equally among the components in the system. That alone, however, did not account for the

FIGURE 1

Example topologies for SPARC T4 and Intel Xeon processors

different characteristics and resource requirements of each thread in a heterogeneous application or workload.

To address this issue, Solaris recently extended its load-balancing mechanism so that a thread's notion of utilization (or required resources) is proportional to its scheduling priority. This allows the scheduler to load-balance lower-priority threads away from where high-priority threads are running, automatically reducing contention for resources. With some simple heuristics, you can safely assume that if a high-priority thread has enough CPU utilization to take advantage of the existing hardware resources, then it should be granted as much access to them as possible.

Automatically identifying which threads or portions of an application should be assigned a higher priority is not a simple task. There is no single characteristic that could allow us to make such decisions for a wide variety of workloads—one could point out several cases where threads with widely different resource requirements are considered critical in the context of their applications. Most critical threads or components, however, are at the top of a dependency relationship in heterogeneous environments. From the startup components of applications to producer/consumer scenarios, any component upon which other parts of the application depend can be considered critical, and they should be assigned a suitably high priority. Such dependency relationships, however, are not easily observable without some previous knowledge of the application architecture.

In Solaris 11, once a performance-critical component or thread is identified, the developer or system administrator has only to place it in the fixed-priority scheduling class at priority 60 or at any real-time priority. The scheduler will then artificially inflate its load according to the underlying platform, attempting to improve its performance by allowing it to execute with more exclusive access to hardware resources. It's important to note that this optimization was implemented to take advantage of the available resources in the system. In other words, if all of the system's logical CPUs are required, no single thread will be forced to wait for the benefit of a single high-priority thread.

This implementation also optimizes differently according to the underlying hardware architecture. On sun4v systems, the scheduler will attempt to provision a performance-critical thread with all of the CPUs sharing an execution pipeline, and a quarter of the CPUs sharing a physical chip on x86 systems. These policies are optimized for both known sources of contention and dynamically resourcing features in their respective platforms. A simple example of the desired behavior would be to have an entire core devoted to a single high-priority thread on a SPARC T4 system, while all the other lower-priority threads share the remaining resources (again, as long as enough idle resources are available in the system).

CONCLUSION

The contemporary landscape of increasing parallelism requires new paradigms. These will affect developers and system administrators at a number of levels in developing new applications and systems. Some are occupied with the considerations of mechanisms at the level of the hardware, virtualization, and operating system. Application developers must have suitable means to designate critical elements of their applications and to interact with the underlying system software to ensure that those elements are given the special resourcing that they require.

ACKNOWLEDGMENTS

I am indebted to Eric Saxe, Jonathan Chew, and Steve Sistare who, among a larger group of colleagues

in the Solaris Kernel Group, were particularly helpful in the development of the ideas presented in this article.

REFERENCES

1. Cantrill, B., Bonwick, J. 2008. Real-world Concurrency. *ACM Queue* 6 (5).
2. Smaalders, B. 2006. Performance Anti-patterns. *ACM Queue* 4 (1).

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

RAFAEL VANONI POLANCZYK is a software developer in the Solaris Kernel Group at Oracle, where he spends most of his time working on the scheduler/dispatcher subsystem. Rafael lives in San Francisco and is originally from Porto Alegre in southern Brazil, where he received a B.Sc. in computer science from UFRGS (Universidade Federal do Rio Grande do Sul).

© 2012 ACM 1542-7730/12/0600 \$10.00