January 2020

Scheduling policy for burst multithreading

N/A

Scheduling policy for burst multithreading

ABSTRACT

Modern computers have many CPU cores, but unless the problem to be solved is highly parallel, these CPU cores cannot be used efficiently: some cores are often left sitting idle while the others perform the bulk of the computation. Even for highly parallel problems, for small procedures, the overhead of requesting a thread to receive the processing request and begin execution is comparable to the time of execution thus resulting in little to no benefits from parallel execution. This disclosure describes techniques to improve the efficiency of communications between threads by reducing overhead. Per the techniques, the runnable threads of a process or a defined subset thereof are scheduled to run simultaneously on the CPU together such that the threads either all run or none of the thread runs.

KEYWORDS

- Parallel programming
- Burst multithreading
- Thread scheduling
- Superscalar execution
- Superscalar scheduling
- Application-level multithreading
- Promises parallel programming
- Spinlocks

BACKGROUND

Modern computers have up to hundreds of CPUs (logical cores that share the same physical chip), but unless the problem to be solved is highly parallel, these CPUs cannot be used efficiently: some of them are left sitting idle while others perform the bulk of the computation. Parallelization approaches typically work at two levels:

- *Superscalar execution*: Superscalar execution parallelizes individual instructions inside a CPU. This parallelization has limits; hence, the tendency is to have many cores rather than more superscalar parallelization.
- *Application-level multithreading*: An example of application-level multithreading patterns is worker-thread pools. Due to the communication overhead between threads, application-level multithreading is efficient when the work items handled by each thread are substantial.

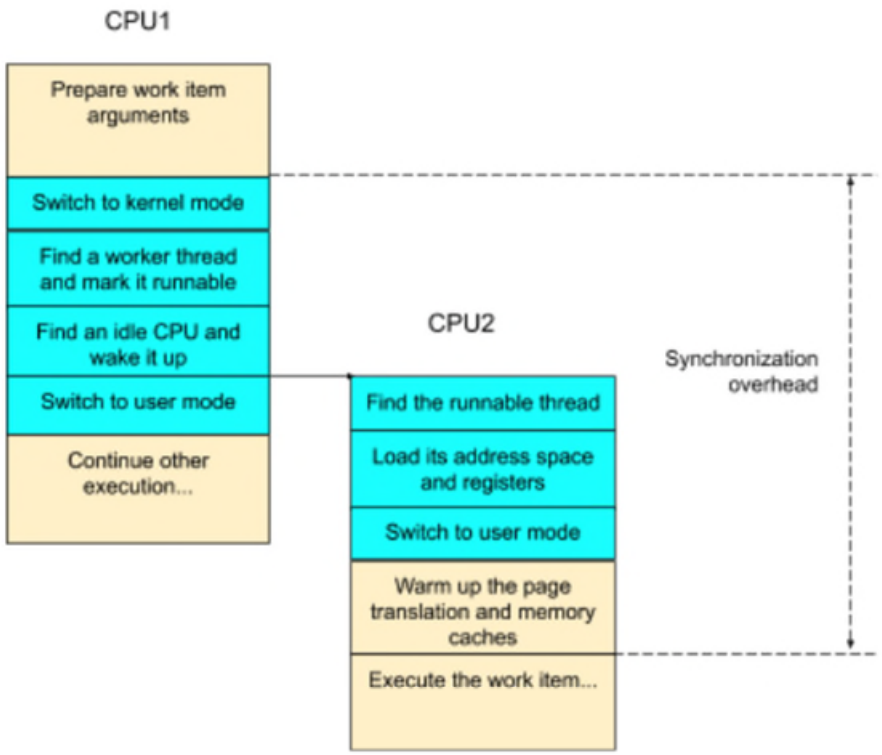

Fig. 1: Scheduling a work item for a worker thread in a worker-thread pool

As illustrated in Fig. 1, scheduling a work item for a worker thread in the worker-thread pool requires going through the operating system (OS) kernel. After the work item arguments are prepared, the code has to do the following:

1. Switch to kernel mode.
2. Find threads awaiting work in the work-synchronization queue.
3. Mark one of these threads as runnable.
4. Find a free CPU and wake it up through inter-CPU communication. The free CPU wakes up in kernel mode.
5. Find the runnable thread, load its virtual address space onto the CPU, and return to the user mode of this thread.

The many steps needed to schedule a work item create a substantial synchronization overhead. Indeed, even for highly parallel problems, the overhead of requesting a thread to receive the processing request and begin execution can be comparable to the time of execution of small procedures. This overhead also depends on whether the stream of requests to the worker threads is contiguous: if it is contiguous then reading the next request from the queue requires only an access or two to the cross-CPU bus, but if it is not contiguous then the sequence starts with a cross-CPU interrupt to an idle CPU, which then causes the OS scheduler to perform such tasks as locating and scheduling the appropriate thread, loading the correct page tables into the CPU, switching the CPU from kernel to user mode, etc.

Besides, the loading of the virtual address space can reset the page translation cache or translation lookaside buffer (TLB), such that when the thread starts running, it is likely to experience TLB misses, thereby consuming more time to load the page cache. Also, the memory

cache might be cold, causing the CPU to spend additional time loading memory values into the cache.

Moreover, to return the result, the same sequence has to be done in the opposite direction, doubling the total overhead. Thus, for efficiency, the work item has to be substantially larger than the overhead involved. Indeed, running a short procedure in a separate parallel thread on a different CPU can be slower due to overhead than just running all the processing on a single CPU.

However, the interactive use of computers tends to impose a bursty load pattern on the CPU: most of the time the CPU does nothing as it awaits human input, but when human input arrives, the computer needs to react as quickly as possible. The interactive computer use case is particularly susceptible to the overhead of short cross-thread requests. Although there are provisions for user-space pause/wake-up/mutexes in some CPU architectures, these come without guarantees of scheduling all relevant threads simultaneously, rendering them difficult to use.

To reduce communications overhead, virtual machine hypervisors frequently schedule all the virtual CPUs of a virtual machine on the physical CPU together. However, scheduling the virtual CPUs of a virtual machine by a hypervisor is driven by the expectation of the guest OS that all the CPUs are continuously available, which is a situation different from a scheduling policy for threads in an OS.

VLIW (very large instruction word) architectures sometimes implement high-granularity parallelism. However, the VLIW implementation of parallelism has certain inherent flaws that permit parallel operation only when CPU counts are low. Some of the problems with VLIW parallelism include:

- A fixed number, e.g., three, of execution streams.
- When an execution stream has nothing to do, its instructions continue being encoded as no-ops, taking up memory and making instruction fetching inefficient.
- Instruction streams cannot execute jumps independently from each other; a jump disrupts the execution order in all three streams.

In the Promises paradigm of parallel programming, a work item is initiated without any guarantees about when exactly it will be executed (this is up to the scheduler), only with a guarantee that the result will be eventually returned when it is needed. A Promise gets initiated well before its result is needed, enabling the thread to start executing it in parallel, well before the result is needed. However, if the execution is not completed (or not started) by the time the result is needed, the caller has to wait for its completion. If the execution was not started when the result is needed, the caller might take the execution upon itself. The Promises paradigm is applicable for large work items in high-level languages, e.g., interpreted languages like JavaScript, but not to the fine-grained granularity of small work items.

DESCRIPTION

This disclosure describes techniques that leverage the opportunity for parallelization between superscalar execution and application-level multithreading. The techniques can be used to parallelize execution of code snippets that are not as small as a few instructions but not as large as a large function. The techniques improve the efficiency of communications between threads by reducing overhead, such that the overhead cost does not overwhelm the efficiency increase arising from parallelization. Per the techniques, all runnable threads of a process (or a defined subset thereof) are scheduled to run simultaneously on the CPU together: they either all

run or none of the threads runs. One example of such a subset is the set of kernel threads used to run a particular tree of fibers, e.g., user-space threads.

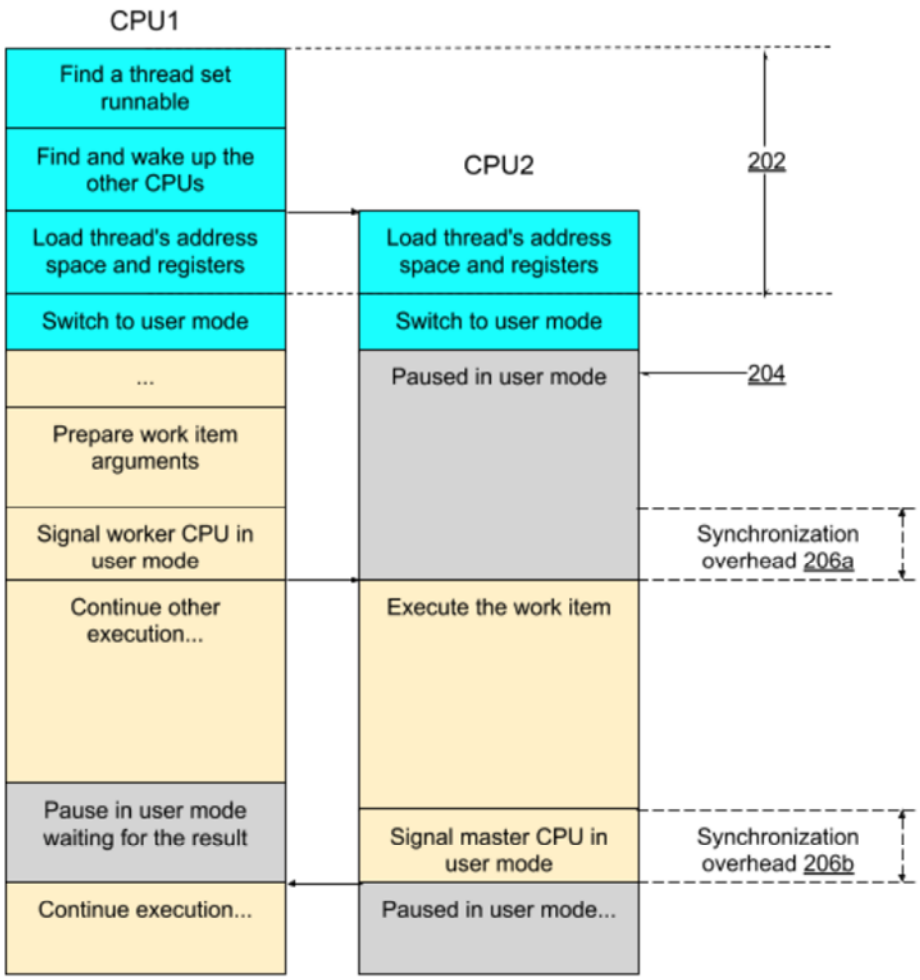

Fig. 2: Improving inter-thread communication efficiency by reducing overhead

Fig. 2 illustrates improving inter-thread communication efficiency by reducing overhead, per techniques of this disclosure. To schedule relatively small work items and the overhead associated with doing so, preparatory work in the OS kernel (202) is done in advance, such that when the work item is ready, it can be started with minimal overhead. This is achieved by scheduling all the threads of a process (or a closely tied subset of the threads of a process) together, even when many threads have nothing to do.

After scheduling, threads that have nothing to do are paused in user mode (204), awaiting work items to be sent their way. This greatly reduces synchronization overhead (206a-b), making relatively small work items efficiently parallelizable. The overhead of the initial setup is amortized across many work items in a time slice. The page cache and memory cache in the CPUs running the worker threads stay warm between executions of the work items. Moreover, a paused worker CPU can snoop on the page cache of the master thread, pre-warming the page cache of the worker with the page cache of the master.

In this manner, small tasks can be scheduled to be executed in parallel threads entirely in user space. Further, the techniques enable a natural preservation of affinity of threads to CPUs, which preserves the CPU caches in a warm state. For example, a kernel thread that is continuously assigned to the same CPU while executing different user-space threads preserves the CPU cache better than the same kernel thread being reassigned to different CPUs every run.

Optionally, the OS can also ensure that the hardware interrupts do not get routed to the CPUs busy with the multithreaded application, providing more reliable guarantees of execution times.

The techniques also apply to other situations, such as the following:

- *User-mode spinlocks:* Spinlocks, where the CPU spins in a tight loop retrying an operation until it succeeds, are normally used in the kernel mode, where there is a guarantee that a thread that currently owns a spinlock will not unexpectedly lose the CPU and force other threads to spin uselessly for a long time. The guarantee of all related threads being scheduled together, per the described techniques, can be used to improve the efficiency of inter-thread communication in massively parallel problems by enabling the use of spinlocks in user mode. In contrast to opportunistic implementations that attempt user-mode spinlock by reverting to

heavyweight, kernel-assisted lock after a limited time of spinning, the techniques enable smaller and simpler straight spinlocks.

- *Mixed-use model:* Not all of the threads of a process have to be scheduled together. Rather, a process can be designed to have one or more subsets of threads with simultaneous scheduling of threads within a subset. The high-*efficiency* inter-thread communication described herein can be used between the threads of a single subset, e.g., to synchronize access for objects that are only accessed by the threads of a given subset.
- *Software implementation of superscalar scheduling:* Rather than implement complex instructions, reduced instruction set CPUs (RISC) implement only a small set of simple instructions in hardware, with *complex* instructions built in software out of the simpler hardware instructions. This gives RISC architectures its superior speed and its applicability to low-energy situations. However, the low-level parallelism of superscalar CPUs, wherein execution units within the CPU are continuously available for scheduling, has given superscalar CPUs a higher performance than RISC.

The intermediate-level parallelism provided by the disclosed techniques can be used as a software-based analog of superscalar scheduling on top of RISC CPUs. When thus applied, the techniques scale with the number of CPUs. Moreover, depending on energy availability, the number of available threads and the number of powered-up CPUs can be adjusted over time. To achieve a software implementation of superscalar scheduling, the compiler defines small parallel work items. If spare CPUs are available, some work items can be executed speculatively, with their results eventually accepted or discarded based on the results of the other operations. Speculative execution is enabled by the use of at least two priority levels, with speculative computations having a lower priority and promoted to a higher priority when they get accepted.

The additional threads can be seen as analogous to execution units inside a superscalar CPU. Code snippets executed by threads in a software-based superscalar scheduling are generally larger than the snippets used inside a superscalar CPU, up to the size of a normal function in a high-level language.

In contrast, e.g., to VLIW implementations of fine-grained parallelism, the techniques ensure that every work item starts with an implicit jump encoded in the arguments and gets executed as an independent instruction stream in a separate CPU that can include jumps, and that the work queue scales transparently with the number of CPUs. Also, in contrast to VLIW implementations of fine-grained parallelism, the techniques have no artificial limitations such as restrictions on the number of execution streams.

CONCLUSION

This disclosure describes techniques to improve the efficiency of communications between threads by reducing overhead. Per the techniques, the runnable threads of a process or a defined subset thereof are scheduled to run simultaneously on the CPU together such that the threads either all run or none of the thread runs.