CMP
United Business Media

MAY 2005

Dr. Dobb's JOURNAL
SOFTWARE TOOLS FOR THE PROFESSIONAL PROGRAMMER

Multithreaded Technology & Multicore Processors

Preparing yourself for next-generation CPUs

CRAIG SZYDLOWSKI

Many software applications are about to be turned upside-down by the transition of CPUs from single to multicore implementations. In new designs, software developers will be tasked with keeping multiple cores busy to avoid leaving performance on the floor. In legacy designs, you will be faced with the challenge of having single-threaded applications run efficiently on multiple cores. Programs will need to serve up code threads that can be dished out to several cores in an efficient manner. Code threading breaks up a software task into subtasks called "threads," which run concurrently and independently.

Threaded code has been the rule in a number of applications for some time, such as storage area networks. Utilizing Hyperthreading Technology from Intel (the company I work for), storage applications deploy concurrent tasks to take advantage of CPU idle time, or CPU underutilized resources, such as when data is retrieved from slow memory. Therefore, tools and expertise are already available to write and optimize threaded code. Operating systems such as Windows XP, QNX, and some distributions of the Linux kernel have been optimized for threading and are ready to support next-generation processors.

Embedded applications are not inherently threaded and may require some software development to prepare for multicore CPUs. In this article, I examine the motivation of CPU vendors to move to multicores, the corresponding software ramifications, and the impact on embedded system developers.

> "All of these CPU implementations require threaded code to fully employ their computing potential"

CPU Architecture Terminology

The terminology to describe various incarnations of CPU architecture is complex. Figure 1 depicts the physical

Craig is an engineer for the Infrastructure Processor Division at Intel. He can be contacted at craig.p.szydlowski@intel.com.

Trend Micro, Inc.
EX1015-1

renditions of three different multithread technologies.

Figure 1(a) shows a dual-processor configuration. Two individual CPUs share a common Processor Side Bus (PSB) that interfaces to a chipset with a memory controller. Each CPU has its own resources to execute programs. These resources include CPU State registers (CS), Interrupt Logic (IL), and an Execution Unit (EU), also called an Arithmetic Logic Unit (ALU).

Figure 1(b) depicts Hyperthreading Technology (HT), which maintains two threads on one physical CPU. Each thread has its own CPU State registers and Interrupt Logic, while the Execution Unit is shared between the two threads. This means the execution unit is time-shared by both threads concurrently, and the execution unit continuously makes progress on both threads. If one thread stalls, perhaps waiting for an operand to be retrieved from memory, the execution unit continues to execute the other thread, resulting in a more fully utilized CPU. Although Hyperthreading Technology is implemented on a single physical CPU, the operating system recognizes two logical processors and schedules tasks to each logical processor.

A dual-core CPU is shown in Figure 1(c). Each core contains its own dedicated processing resources similar to an individual CPU, except for the Processor Side Bus, which may be shared between the two cores.

All of these CPU implementations require threaded code to fully employ their computing potential. In the future, the dual-core CPU model will be extended to quad-core, containing four cores on a single piece of silicon.

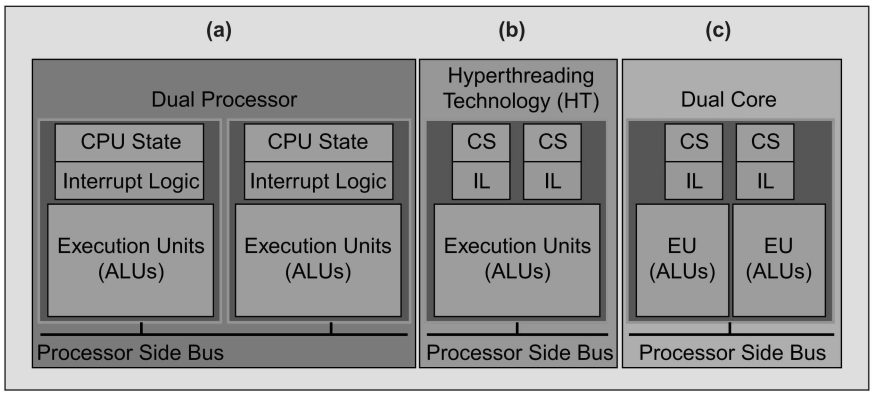


Figure 1: *Three multithread technologies.*

Why the Move to Dual-Core?

Ever increasing clock speed is creating a power dissipation problem for semiconductor manufacturers. The faster clock speeds typically require additional transistors and higher input voltages, resulting in greater power consumption.

The latest semiconductor technologies support more and more transistors. The downside is that every transistor leaks a small amount of current, the sum of which is problematic.

Instead of pushing chips to run faster, CPU designers are adding resources, such as more cores and more cache to provide comparable or better performance at lower power. Additional transistors are being leveraged to create more diverse capability, such as virtualization technology or security features as opposed to driving to higher clock speeds. These diverse capabilities ultimately bring more performance to embedded applications within a lower power budget. Dual-core CPUs, for example, can be clocked at slower speeds and supplied with lower voltage to yield greater performance per watt.

Parallelism and Its Software Impact

Multicore processor implementation will have a significant impact on embedded applications. To take advantage of multicore CPUs, programs require some level of migration to a threaded software model and necessitate incremental validation and performance tuning. There are kernel or system threads managed by the operating system and user threads maintained by programmers. Here I focus on user threads.

You should choose a threaded programming model that suits the parallelism inherent to the application. When there are a number of independent tasks that run in parallel, the application is suited to functional decomposition. Explicit threading is usually best for functional decomposition. When there is a large set of independent data that must be processed through the same operation, the application is suited to data decomposition. Compiler-directed methods, such as OpenMP (http://www.openmp.org/), are designed to express data parallelism. The following example describes explicit threading and compiler-directed methods in more detail.

To exploit multicore CPUs, you identify the parallelism within your programs and create threads to run multiple tasks concurrently. The vision-inspection system in Figure 2 illustrates the concept of threading with respect to functional and data parallelism. You must also decide upon which threading models to implement — explicit threading or compiler-directed threading.

The vision-inspection system in Figure 2 measures the size and placement of leads onto a semiconductor package. The system runs several concurrent function tasks, such as interfacing to a human, controlling a conveyer belt, capturing images of the leads, processing the lead images, and detecting defects and transferring the data to a storage area network. These tasks represent functional parallelism because they run at the same time, execute as individual threads, and are relatively independent. These tasks are asynchronous to each other, meaning they don't start and end at the same time.

The advantage of threading these functional tasks is that the inspection application doesn't lock up when other tasks or functions run, so the machine operator, for example, experiences a more responsive application.

The processing of the semiconductor package images is well-suited to data parallelism because the same algorithm is run on a large number of data elements. In this case, the defect detection algorithm processes arrays of pixels by

looping and applying the same inspection operation to independent sets of pixels. Each set of pixels is processed by its own thread.

For either functional or data parallelism, you can write explicit threads to instruct the operating system to run these tasks concurrently. An explicit thread is purposely coded instructions using thread libraries such as Pthreads or Win32 threading APIs. You are responsible for creating threads manually by encapsulating independent work into functions that are mapped to threads. Like memory allocation, thread creation must also be validated by you.

Although explicit threads are general purpose and powerful, their complexity may make compiler-directed threading a more appealing alternative. An example of compiler-directed threading is OpenMP, which is an industry standard set of compiler directives. In OpenMP, you use pragmas to describe parallelism to the compiler; for example:

```
#pragma omp parallel for
                    private(pixelX,pixelY)
  for (pixelX = 0; pixelX <
                    imageHeight; pixelX++)
  {
     for (pixelY = 0; pixelY <
                    imageWidth; pixelY++)
     {
   newImage[pixelX,pixelY] =
     ProcessPixel (pixelX, pixelY, image);
     }
  }
```

The *pragma omp* says this is an opportunity for OpenMP parallelism. The *parallel* key word tells the compiler to create threads. The *for* key word tells the compiler the iterations of the next *for* loop will be divided amongst those threads. The *private* clause lists variables that need to be kept private for each thread to avoid race conditions and data corruption.

The compiler creates the spawned threads as in Figure 3. Notice the spawned threads are all created and retired at the same time, somewhat resembling the tines of a fork. There is an explicit parent-child relationship that is not necessary with threaded libraries. This is called a "Fork-Join" model and is a required characteristic for OpenMP parallelism. OpenMP pragmas are less general than threaded libraries, but they are less complex because the compiler creates the underlying parallel

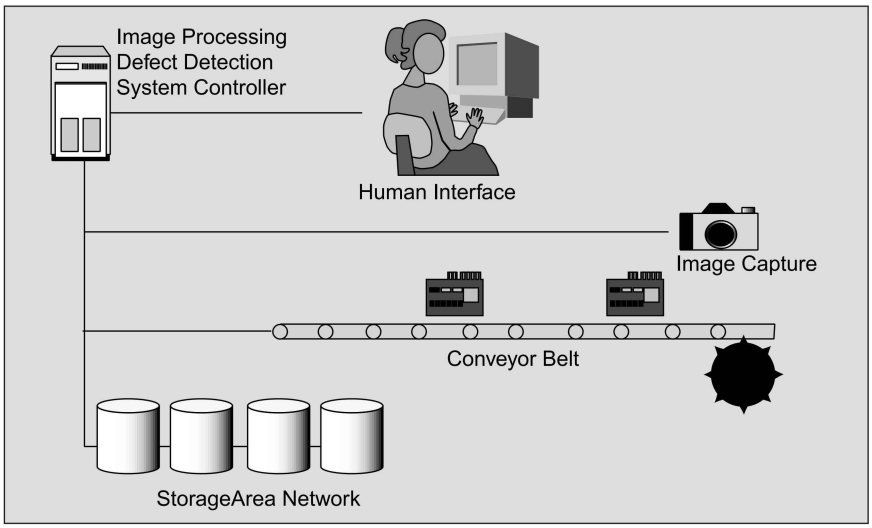


Figure 2: *Typical vision-inspection system.*

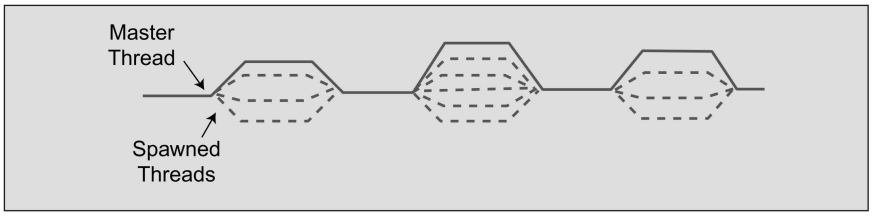


Figure 3: *Fork-Join model.*

code for the multiple threads. OpenMP is supported by various compilers allowing the threaded code to be transportable, whereas threaded libraries typically have allegiance to specific operating systems.

Parallelism Debug

Whether threads are created explicitly, by compiler directive, or by any other method, they need to be tested to ensure no race conditions exist. With a race condition, you have mistakenly assumed a particular order of execution, but didn't guarantee that order. In embedded applications, processes are often asynchronous, which means a bug may be dormant during validation testing and permits the code to work nearly all the time.

A race condition may be caused by a storage conflict. Two threads could be overwriting a particular memory location or a thread may presume another thread completed its work on a particular variable, leading to the use of corrupt data. Access to common data must be synchronized to avoid data loss. Synchronization can be implemented with a simple status word to indicate the state of the data called a "semaphore." A thread takes control of the data by writing "0" to the status word, whereas writing "1" to the status word releases control, allowing another thread to access the variable. As embedded applications are often interrupt driven, it may be useful to implement a protected read-modify-write sequence to guarantee a thread's operations on a variable are not disturbed by another process such as an interrupt service routine.

There are sophisticated tools available to test for race conditions. The Intel Thread Checker (http://www.intel.com/ids/) is an automated runtime debugger that checks for storage conflicts and looks for places where threads may lock or stall. It identifies memory locations that are accessed by one thread, followed by an unprotected access by another thread, which exposes the program to data corruption. The Thread Checker is a dynamic analysis tool and is, therefore, dataset dependent. As such, if the dataset does not exercise certain program flows, the tool is not capable of checking that code portion. For embedded applications, it is important to create a dataset that simulates the relevant asynchronous processes.

Finding race conditions can be very dif-

ficult and time consuming. Thread Checker can easily find these conflicts, even when the conflict is generated by code instances in different call stacks and many thousands of lines apart.

Performance Tuning

Once the code has been tested and verified to be executing correctly, performance optimization may begin. Performance optimization should be limited to the critical path of execution. There is little return for tuning code with no impact to overall system performance.

To maximize the performance of multicore CPUs, it is often necessary to ensure that the workload is balanced between the cores. Load imbalance will limit parallel inefficiency and scalability because some processor resources will be idle.

Synchronization can also limit performance by creating bottlenecks and overhead. Although synchronization helps guarantee data integrity, it serializes the program flow. Synchronization requires some threads to wait on other threads before the program flow can proceed, resulting in idle processor resources.

To assist performance tuning, the Intel Thread Profiler (http://www.intel.com/ids/) lets you check load balance, lock contention, synchronization bottlenecks, and parallel overhead. This tool can drill down to source code for threads created by OpenMP or thread libraries. The profiler identifies the critical path in the program and indicates the processor utilization by thread. You can view the threads in the critical path as well as the CPU time spent in each thread.

Impact of Multicore CPUs on Embedded Systems

Hardware and software developers of embedded systems will be impacted by the move to multicore CPUs. Hopefully, board designers will find multicore CPUs alleviate the thermal issues of today's high-performance processors, while providing comparable performance. Programmers may need to adapt to new programming models that include threaded software. Although creating, checking, and tuning threads may initially be arduous, it will provide you with more control over the resources of the CPU and possibly decrease program latencies. Those developing real-time systems can partition work amongst multiple cores and assign priorities in order to get critical tasks completed faster.

Software developers who fail to prepare for the transition to multicore CPUs may either get pigeon-holed onto older CPUs or risk performance issues from unoptimized code.

Many tools are available to help the transition to threaded code. Through multithreaded capabilities such as Hyper-threading Technology, many developers already have experience with specialized tools and threaded programming models. This background and code development will provide an immediate payback when these applications run on dual-core CPUs.

Multicore has also raised the question of software licensing and the associated costs that customers will have to pay. Some software vendors have considered charging license fees on a per-core basis, charging more for dual- or multicore systems. Against this tide, Microsoft has announced that its software will be licensed on a per-processor package basis. This means only a single license is needed, regardless of how many cores are contained within the processor.

The groundwork is being laid for the transition to multicore CPUs in 2005. Tools are available to help you develop efficient and reliable threaded code. Embedded application providers should plan their move to threaded programming models to fully utilize the performance of next generation multicore CPUs.

DDJ

Copyright© 2005 by CMP Media LLC, 600 Community Drive, Manhasset, NY 11030. Reprinted from Dr.Dobb's Journal (http://www.ddj.com) with permission. 5789