Multi-Threading in the .NET* Environment

by Matt Gillespie

April 2005

The information contained in this document is provided for informational purposes only and represents the current view of Intel Corporation Intel on the date of publication. Intel makes no commitment to update the information contained in this document, and Intel reserves the right to make changes at any time, without notice.

DISCLAIMER. THIS DOCUMENT AND ALL INFORMATION CONTAINED HEREIN IS PROVIDED AS IS. INTEL MAKES NO REPRESENTATIONS OF ANY KIND WITH RESPECT TO PRODUCTS REFERENCED HEREIN, WHETHER SUCH PRODUCTS ARE THOSE OF INTEL OR THIRD PARTIES. INTEL EXPRESSLY DISCLAIMS ANY AND ALL WARRANTIES, IMPLIED OR EXPRESS, INCLUDING WITHOUT LIMITATION, ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, AND ANY WARRANTY ARISING OUT OF THE INFORMATION CONTAINED HEREIN, INCLUDING WITHOUT LIMITATION, ANY PRODUCTS, SPECIFICATIONS, OR OTHER MATERIALS REFERENCED HEREIN. INTEL DOES NOT WARRANT THAT THIS DOCUMENT OR THE INFORMATION CONTAINED HEREIN IS FREE FROM ERRORS, OR THAT ANY PRODUCTS OR OTHER TECHNOLOGY DEVELOPED IN CONFORMANCE WITH THIS DOCUMENT WILL PERFORM IN THE INTENDED MANNER, OR WILL BE FREE FROM INFRINGEMENT OF THIRD PARTY PROPRIETARY RIGHTS, AND INTEL DISCLAIMS ALL LIABILITY THEREFOR.

INTEL DOES NOT WARRANT THAT ANY PRODUCT REFERENCED HEREIN OR ANY PRODUCT OR TECHNOLOGY DEVELOPED IN RELIANCE UPON THIS DOCUMENT, IN WHOLE OR IN PART, WILL BE SUFFICIENT, ACCURATE, RELIABLE, COMPLETE, FREE FROM DEFECTS OR SAFE FOR ITS INTENDED PURPOSE, AND HEREBY DISCLAIMS ALL LIABILITIES THEREFOR. ANY PERSON MAKING, USING OR SELLING SUCH PRODUCT OR TECHNOLOGY DOES SO AT HIS OR HER OWN RISK.

Licenses may be required. Intel and others may have patents or pending patent applications, trademarks, copyrights or other intellectual proprietary rights covering subject matter contained or described in this document. No license, express, implied, by estoppels or otherwise, to any intellectual property rights of Intel or any other party is granted herein. It is your responsibility to seek licenses for such intellectual property rights from Intel and others where appropriate.

Limited License Grant. Intel hereby grants you a limited copyright license to copy this document for your use and internal distribution only. You may not distribute this document externally, in whole or in part, to any other person or entity.

LIMITED LIABILITY. IN NO EVENT SHALL INTEL HAVE ANY LIABILITY TO YOU OR TO ANY OTHER THIRD PARTY, FOR ANY LOST PROFITS, LOST DATA, LOSS OF USE OR COSTS OF PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES, OR FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF YOUR USE OF THIS DOCUMENT OR RELIANCE UPON THE INFORMATION CONTAINED HEREIN, UNDER ANY CAUSE OF ACTION OR THEORY OF LIABILITY, AND IRRESPECTIVE OF WHETHER INTEL HAS ADVANCE NOTICE OF THE POSSIBILITY OF SUCH DAMAGES. THESE LIMITATIONS SHALL APPLY NOTWITHSTANDING THE FAILURE OF THE ESSENTIAL PURPOSE OF ANY LIMITED REMEDY.

Intel, the Intel logo, Pentium, Intel Xeon, and VTune are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

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

Copyright © 2005 Intel Corporation

Introduction

Building multi-threading into .NET* applications is a powerful tool for the developer. It empowers applications that demand high scalability, such as enterprise applications, as well as desktop applications that need to process more than one task at the same time. In order to enable multi-threading technology, Intel developed Hyper-Threading Technology (HT Technology) for the Pentium® 4 processor on the desktop and the Intel® Xeon™ processor for workstations and servers. These platforms make a particularly good choice for the deployment of threaded .NET solutions.

The coming trend of parallelism in hardware is that of multi-core processors, which will make multi-threading even more important. Intel has announced planes to introduce the first versions of both the Itanium® processor and the Pentium 4 processor that have multiple processor cores on a single die in 2005. These introductions will be followed in 2006 by the first multi-core Intel Xeon processors and Intel Xeon processors MP in 2006. As these and other advances make hardware platforms ever more parallel in nature, the stage is set for software threading to deliver ever-larger performance gains. Software makers that gear up for this advance now, stand to gain a competitive advantage as this trend continues.

This article provides decision makers and developers with information about the ways in which threading improves the performance of .NET applications, as well as how threaded .NET applications take advantage of the hardware features of Intel® architecture. It also gives examples of threading pitfalls in .NET and ways of resolving them. The article closes with a brief introduction to threading-related Intel® software-development tools for the .NET developer.

Using Threads to Increase Performance

Operating systems can allocate processor resources to individual threads, each of which executes application code. Those threads are each assigned slices of processor time, and they take turns with the other threads, in round-robin fashion, to carry out useful work for the user. This capability enables a number of performance advantages, including the ability to prioritize work and to allow background tasks to take place, for instance, while a user interface is idle.

Each thread must maintain a certain overhead associated with its context relative to the larger application, such as its priority and pointers to associate it with its host process and its set of CPU registers. Processor resources are also required to suspend and resume thread execution, so there is an inherent tradeoff between the advantages and the overhead associated with creating additional threads. It is also vital to avoid conflicts between threads for the same piece of data.

Since the slices of processor time allocated to each thread are very small, the operating system's rapid switching between threads creates the effect of simultaneous execution on a single processor, although in reality, instructions are being retired on only one thread at any given time. Once the executing thread's allocated time slice expires, the operating system stores the context information of that thread and suspends execution. It then reloads the context information for the next thread, and resumes execution on that next thread.

HT Technology, which is supported by Intel Xeon processors and many Pentium 4 processors, enables a single physical processor to expose two logical processors to the operating system. Thus,

when a thread must wait for data to be fetched from memory, the second logical processor is able to continue doing useful work. This capability builds further upon the performance increases that are available through threading, by allowing two threads to run concurrently on each physical processor.

A detailed inquiry into threading best practices is beyond the scope of this article, although ample resources are available from the Intel® Developer Services Threading Developer Center; the Threading Knowledge Base is particularly useful for obtaining solutions to specific threading challenges quickly.

Threading should be used whenever you can make your program more responsive, efficient or usable by assigning specific functionality to a distinct thread. Of course, the corollary is to employ multi-threading only when it is needed, as overuse can lead to development and performance issues. A few examples of places where multi-threading would be advantageous are explored below:

- **Multiple threads can help improve user-interfaces responsiveness and foster efficient communication with other systems.** By allowing a user interface to run in a separate process simultaneously with background processing tasks, users are able to continue input tasks and other interactions, even while the application is performing heavy computations in the background.
- **Multi-threading can reduce the performance impacts associated with long-running processes.** Since operations that take a long time to complete might otherwise block other tasks from executing, generating multiple threads can provide improved resource sharing. For example, if an inventory application takes a long time to generate a weekly report based on a complex database query, you would not want incoming orders in an e-commerce environment slowed down by that operation.
- **By aiding in the prioritization of tasks, threaded code can ensure that time-critical tasks are handled appropriately.** Assigning different threads appropriate priority levels can help to ensure that tasks that require low latencies are not slowed down by lower-priority tasks. For instance, in a Voice over IP application, voice-decoding and encoding operations must run at a high priority to ensure real-time fidelity, while other tasks such as repainting the screen should run at a lower priority.

The .NET Common Language Runtime (CLR) provides rich threading support, including excellent support for HT Technology.

How Intel Architecture Benefits Multi-Threaded .NET Applications

Both Intel Xeon processors and Pentium 4 processors provide a solid foundation upon which to deploy .NET solutions. The .NET Framework is optimized for both platforms, enabled by a strong tradition of collaboration between Microsoft and Intel that ensures the strength of ongoing efforts on the parts of both parties to make sure that the products of each company takes best advantage of the other's. Developers can therefore have a very high degree of confidence in the correctness and optimization of the threading implementation of the .NET Framework for Intel architecture.

One strong example of the benefits of that collaboration is the .NET platform's rich optimization for HT Technology, which allows the developer to take advantage of the technology with no specific

changes to the code beyond solid general threading fundamentals. This optimization at the Framework level clearly simplifies the developer's task in tuning an application's threading behavior.

Intel has announced that future plans for both the Pentium 4 processor and the Intel Xeon processor include the introduction of multi-core versions. By placing multiple processor cores on a single processor die, Intel expects to generate extremely high performance by decreasing the latency in communication between the cores, relative to completely separate processors. This increase in performance will add further benefit to threaded .NET applications, beyond that which is available today.

Avoiding Deadlocks in Threaded .NET Applications

One of the most common pitfalls associated with threaded .NET code is the deadlock. This situation occurs when each of the two threads is waiting for the other to complete some process in order to proceed. At the left side of the following figure, Thread X places an exclusive lock on Resource A, while Thread Y places an exclusive lock on Resource B. At the right side of the figure, they have acquired those locks. Meanwhile, Thread X has attempted to place an exclusive lock on Resource B, and Thread Y has attempted to place an exclusive lock on Resource A. Since neither of those resources are available, if neither process can proceed without them, both threads are deadlocked.

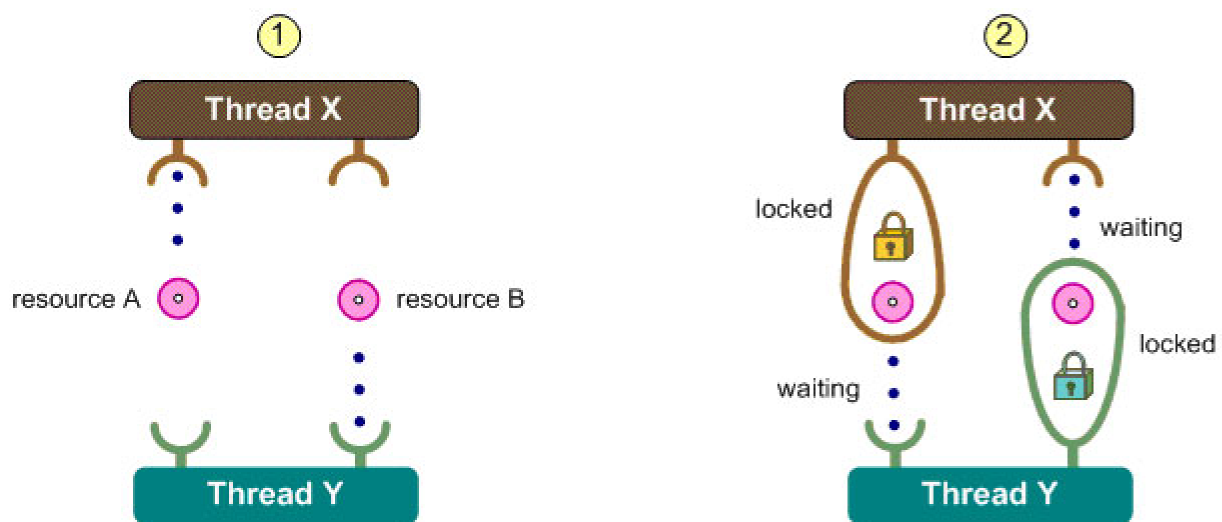

Figure 1. Threads X and Y become deadlocked as each waits for the other to release a locked resource.

This representation is a highly simplified instance of the situation where each thread is waiting for the other to release a resource before proceeding. Neither thread can perform any work, and therefore, the processor resources that each of the threads has assigned to it remain idle for the duration of the deadlock. The .NET Framework provides a number of useful methods that can be used to prevent deadlocks, including the Monitor.TryEnter method, which is documented in the method's MSDN .NET Framework Class Library entry*.

Briefly, Monitor.TryEnter allows the developer to have the Framework try to acquire an exclusive lock on a specific resource for a specified period of time. The method returns a logical True or False

value that indicates whether or not the exclusive lock was obtained. Using this technique, one can create a timeout that determines whether acquiring an exclusive lock on a resource is possible, and if not, logic can be built into the code to work around that unavailability, rather than simply waiting idle until the resource becomes available.

Avoiding Race Conditions

A race condition is another common threading pitfall. In this circumstance, multiple threads will each operate on a particular block of code, but the overall outcome of the execution is dependent upon which thread reaches the block of code first. For example, consider the case of two processes that each increment the value X by 1, which consists of three distinct operations:

1. Load the value of the variable **X** into a register.
2. Add 1 to the value in the register.
3. Write the contents of the register back into the variable **X**.

In this highly simplified example, it is clear that if the two processes each operate one after the other, the final value of X after both processes complete will be equal to the initial value plus 2. Thus, if the initial value of X is 5, the first process will increment it to 6, and then the second process will increment it to 7. Another possible outcome, however, is that the first process could complete the first two steps outlined above, and then that process could be preempted by the second process. The second process would read the value of X as 5, increment it to 6, and then write it back to the variable X. Once the first process resumes, it would continue where it left off, writing the 6 back as the value of X. Thus, in the first case, the code generates a final value of 7 for X, and in the second case, the same code generates a final value of 6 for X. Which result would be generated by any particular execution of the code could be unpredictable.

The .NET Framework provides the Increment method (along with its companion Decrement method) of the Interlocked class to prevent this particular type of race condition. This class is documented in its MSDN .NET Framework Class Library entry*. These methods perform the three steps required to increment or decrement a variable as a single operation, preventing a thread performing it from being preempted before it writes the value back to memory. Related issues and other types of race conditions are delineated in the Synchronizing Data for Multithreading* section of the .NET Framework Developer's Guide.

The general best practice for developers with regard to potential race conditions in threaded code is that one must always consider what would happen if the thread executing a passage were preempted before completing that execution. As the forgoing example illustrates, issues of preemption can also occur when a thread is preempted between the machine-code operations that make up even a single line of higher-level programming code.

Plug-Ins to Simplify Threading

Intel provides the Intel® Thread Checker and Thread Profiler as plug-ins that work with the Intel® VTune™ Performance Analyzer environment to improve the rate of success that developers enjoy in creating threaded code. Under .NET, Intel Thread Checker integrates with the Microsoft Visual Studio* .NET environment (including older versions of the Visual Studio compilers) to identify threading errors such as deadlocks and race conditions in applications, supporting source

instrumentation and allowing Intel Thread Checker to drill down to the specific variable that caused the error. In fact, even if the source code is not available, the tool supports binary instrumentation that can identify the line of source code associated with the error, allowing the developer to identify the specific variable(s) through external debugging methods.

Once the application is instrumented, the developer runs it in conjunction with Intel Thread Checker to monitor the application and generate diagnostic reports. Those reports give rich detail about each error, including the interactions with other threads that may underlie them. Analysis of these reports helps developers to efficiently resolve the errors by means of on-board diagnostics, error detection and classification, and graphical representations of errors that allow you to categorize and manipulate errors, sorting them against each other and against specific portions of source code for dynamic analysis.

It is important to remember that, even in the absence of actual errors, incorrect threading practices can dramatically reduce the performance of an application, relative to an unthreaded version. In addition to resolving threading runtime errors, therefore, it is also vital to find those places in code where threading has created inefficiencies. Moreover, detecting and resolving those issues can be a very complex undertaking, and so it is very important to have the correct tools to help.

Intel Thread Profiler integrates with the Visual Studio .NET environment and Intel VTune Performance Analyzer to analyze the threading performance of applications in real time during execution. Key aspects of threading behavior that the developer will want to look at in order to improve performance include synchronization/threading overhead and load balancing among threads. Thread Profiler is extremely helpful in identifying and resolving both types of issues.

Thread Profiler performs critical-path analysis to generate runtime statistics that the developer can view in a number of ways, organizing the data by thread or by region of code. By drilling down to specific bodies of performance data for a given application, the developer is able to analyze the behavior of a specific piece of code. In this manner, one can identify hotspots in serial regions, parallel regions, and critical sections, as well as finding synchronization tasks that have a detrimental impact on performance.

The Intel VTune analyzer environment equipped with Thread Profiler provides specific, targeted tuning advice for improving the performance associated with each threading issue, such as synchronization delays, stalled threads, and excessive blocking time. It also provides analysis of the utilization of processor resources at specific parts of program execution, since either under-utilization or over-utilization indicates an issue that should be addressed. Most important, Thread Profiler provides intelligent analysis that enables the developer to determine which tuning tasks to prioritize, in order to meet performance goals with the least amount of tuning effort.

Conclusion

Multi-threading is a powerful means of getting highest performance out of enterprise applications, and its role will continue to grow as hardware platforms continue to become more parallel in nature. The introduction of multi-core processors on all of Intel's desktop and server processor lines by 2006 shows an important aspect of where the industry is heading, and threading your applications now places you in a good position to take advantage of this trend as the industry moves forward. The next major phase of this advance begins in 2005 with the first multi-core Itanium processor and multi-core Pentium 4 processor, which will run the next release of many software products.

Both Intel architecture and the .NET Framework provide a number of technologies that enable users to put threading to work efficiently and safely. Nevertheless, the benefits of threading are accompanied by dangers, if the technologies are not implemented correctly, and so Intel has provided software-development tools that help to ensure optimal utilization of threading in your applications. Those tools can help you to manage the complexity of threading well, making it easier to achieve success.

Developers should take the time to become familiar with the threading support that is built into the .NET Framework, in order to build the most robust code possible. Familiarity with the hardware features such as HT Technology in the latest Intel processors also helps to support high threading performance, and there is no substitute for completing your development environment with low-cost Intel threading tools that will allow you to detect threading errors and simplify the tuning process for threaded applications.

About the Author

Matt Gillespie is an independent technical author and editor working out of the Chicago area and specializing in emerging hardware and software technologies. Before going into business for himself, Matt developed training for software developers at Intel Corporation and worked in Internet Technical Services at California Federal Bank. He spent his early years as a writer and editor in the fields of financial publishing and neuroscience. You can reach him at spanningtree@comcast.net.

Related Links

- Intel Developer Services .NET Developer Center provides a wide array of information and resources for the .NET developer, including techniques and training for optimizing .NET applications.
- Intel Developer Services Threading Knowledge Base provides concise answers to common threading issues in an efficient-to-use Challenge and Solution format.
- Intel® Developer Services Threading Forum offers in depth discussions with Intel experts and your peers on practical threading challenges.
- MSDN .NET Framework Developer's Guide: Threading* introduces managed threading concepts and the use of managed threading for the .NET Framework.
- The Intel® Software College provides a one-stop shop at Intel for training developers on leading-edge software-development technologies. Training consists of online and instructor-led courses covering all Intel architectures, platforms, tools, and technologies.

intel®