Getting the Most from your Middleware

Introduction

The Intel® Laptop Gaming Technology Development Kit (Intel® Laptop Gaming TDK) [1] enables game developers and game engine developers to design and develop laptop-aware games. The Intel Laptop Gaming TDK does this by providing functionalities that dynamically detect changes in system-state information such as power, network and processor. Using this information, game developers can alter game behavior to maximize game play on laptops. Game developers can also use the Intel Laptop Gaming TDK to receive notifications of platform events to achieve maximum game play. In this context, “maximum game play” means utilizing mobility features in a game in a way that reduces power consumption, which in turn allows for maximum battery life while playing the game. The Intel Laptop Gaming TDK also allows game developers to monitor system status, such as remaining battery life, allowing them to alert the game player to the situation.

Oversubscription is bad for you

Modern game development increasingly uses middleware products to stay within time and money budgets. This is a sensible approach, but it puts game developers at the mercy of middleware designers to ensure acceptable performance for key game functions. Knowing this, many middleware developers have begun to modify their products to enable parallel execution of middleware work. But, as with all parallel programming, there are performance pitfalls to avoid. The key challenge in this scenario is to avoid oversubscription: the performance penalty that occurs when more threads are created than the hardware resources can handle. Although it’s a problem for all parallel programs, oversubscription is particularly dangerous for games that compute each frame in a fixed time budget. If the game’s critical path takes longer than this time budget then a frame could get dropped.

When integrating parallel middleware into a game, care must be taken to ensure that the middleware’s parallel approach doesn’t compete with the game’s computation, or with other middleware already in use. If the game and each middleware create threads without regard to overall hardware resources, this conflict is almost unavoidable and results in an oversubscription performance penalty. Thoughtful design of the middleware API can help prevent this problem from occurring, but it is equally incumbent on the game developer to use the API appropriately.
Three strategies for avoiding the oversubscription penalty

Middleware packages should provide at least one API approach for handling parallelism. Multiple API approaches will help the middleware adapt more easily to different game architectures. Three different approaches - thread parceling, work pulling, and work pushing - can be used to avoid oversubscription, each with advantages and disadvantages.

**Thread parceling**

Thread parceling means specifying how many threads/cores to devote to the host and to each middleware. This is a very common API design in current threaded middleware, and it is typically sufficient when both of the following are true:

1. The game is designed to run on a single homogenous gaming platform.
2. The game’s calculation frame-to-frame is relatively constant.

Realistically, most commercial games target multiple gaming platforms, and/or the diverse PC platform. Determining a thread parceling scheme for each CPU core topology is possible, but tedious. Also, games typically vary their computational load dramatically throughout the game. A sudden surge in middleware processing can cause frame rates to drop in a thread parceling scheme. As game complexity increases, and gaming platforms continue to appear and diversify, thread parceling will become increasingly difficult to parameterize successfully.

**Work pulling**

Work pulling is a way for host and middleware work to be executed on the same threads. In this scheme the game is the only code that spawns any threads. The middleware internally decomposes its work into tasks that can be invoked individually by the game. In practice, the game will schedule this work to execute on its own threads which will also be executing game work. The game threads must be prepared with any thread-specific state necessary to run middleware work. Since the game is the only code to spawn threads, accidental oversubscription is not possible.

![Figure 1. Work Pulling Sequence Diagram](image-url)
Work pushing

Work pushing is another way for host and middleware work to share threads. This method requires the middleware to define an abstract interface for scheduling work, which the game provides as an instantiated object. Each frame, the middleware pushes its work onto the scheduling object, and the game ensures that work is executed by its own threads. This is more complicated than the work pulling approach, but it has the advantage of more flexibility. Instead of the middleware internally subdividing its work into an arbitrary number of chunks, it can schedule the work as one splittable task which will be divided as needed by the game threads. The splittable task will remain in one piece if only one thread is able to work on it, or split into more pieces if more threads are available. Like the work pulling scheme, this approach avoids oversubscription since the game is the only code that spawns threads.

The StressTest demo shows relative performance of different API designs

We created some samples that demonstrate the oversubscription penalty and some strategies for avoiding that performance penalty. The StressTest demo simulates the activity of a threaded game and a threaded middleware package by having each execute some simple computationally intense work each frame. Serial, oversubscribed, work pulling, and work pushing API usages are compared against each other. On n-core, the oversubscribed case uses n threads for the game, and n-1 threads for the middleware, for a total of 2n-1 threads.

Table 1 shows the timings in seconds of the four different API designs when run on 2 to 8 cores of an Intel® Core™ i7 processor. The same data is displayed graphically below.

The timing data from the demo shows that the work pulling and work pushing APIs perform nearly identically. Both of these APIs are between 20% and 40%. Faster than the oversubscribed case.
**StressTest:** time in seconds of different API designs, on 2 to 8 cores of an Intel® Core i7™ processor.

**StressTest:** speedup of different API designs versus serial performance, on 2 to 8 cores of an Intel® Core i7™ processor.
Note that the work pushing and work pulling results are so similar that the work pushing line is mostly obscured by the work pulling line. The work pulling line has been made thinner to show the work pushing line behind it. These charts make it clear that both the work pulling and work pushing schemes consistently outperform the oversubscribed case. There is no performance penalty for using either the work pushing or work pulling APIs, relative to the oversubscription case.

The GameTest demo shows how oversubscription impacts frame rate

The StressTest demo simulated the case where both the game and the middleware have plenty of parallel work to do. Real games usually have lower overall CPU utilization, with some portion of the computation being tightly bound to a time budget. Oversubscription can cause the time budget to be overrun. The GameTest demo explores this case by simulating a game with 70% utilized CPU, with 50% used by the main game code, and 20% used by the middleware. The game has a time critical piece of code that is calibrated to occupy most of the time needed to generate a frame at 30 frames per second. If that code takes too long to execute, the frame draw will be skipped, resulting in an uneven framerate. The results are shown in Table 2.

The upper part of Table 2 shows the time budget in milliseconds for drawing at 30 frames per second alongside the average frame time of the oversubscribed case and of the work pulling API. Work pulling has a clear advantage on both 4-core and 8-core, staying within the time budget. This data is shown graphically below.
**GameTest:** average frame times in milliseconds, compared to time budget, with game at 50% CPU utilization and middleware at 20% CPU utilization

**GameTest:** percentage of missed frames, with game at 50% CPU utilization and middleware at 20% CPU utilization
The average frame time is not the only interesting metric. Every time a frame is missed, stuttering and choppiness are created in the visual experience. The lower part of Table 2 shows the number of missed frames for the two techniques, which again clearly favors the work pulling API on 4-core and 8-core.

**Threaded middleware requires thoughtful design and use**

As the middleware market matures and the installed base of multi-core gaming machines grows, more and more middleware will become parallelized for performance. It is important that the method of parallelizing middleware evolves with the market; otherwise the promise of parallel performance will go unrealized. The demo created for this article demonstrates that an oversubscription penalty can reduce performance by as much as 40%, even on a 2-core system. In games with a strict time budget, the oversubscription penalty can dramatically increase the number of missed frames. This performance penalty is relatively easy to avoid with thoughtful API design and use. Middleware developers: ensure that your product’s API provides your customer with a method of avoiding oversubscription penalties. Game developers: when evaluating middleware for your game, be sure to consider how well the middleware implements its parallel API. Your game needs all the performance it can get, and there’s no reason to allow an oversubscription penalty to reduce that performance.

**Further reading and resources**

The demo code referenced in this article is available at TBD.


Stay current on the lastest software developments in Visual Computing at:

[http://visualadrenaline.intel.com](http://visualadrenaline.intel.com)