Developing Platform Consistent Multithreaded Applications: Memory Management

March 2003

Copyright © Intel Corporation 2003
Terms of Use

THIS DOCUMENT IS PROVIDED "AS IS" WITH NO WARRANTIES WHATSOEVER, INCLUDING ANY
WARRANTY OF MERCHANTABILITY, FITNESS FOR ANY PARTICULAR PURPOSE, OR ANY
WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.

Information in this document is provided in connection with Intel products. No license, express or implied, by
estoppel or otherwise, to any intellectual property rights is granted by this document or by the sale of Intel
products. EXCEPT AS PROVIDED IN INTEL'S TERMS AND CONDITIONS OF SALE FOR SUCH
PRODUCTS, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS
OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS INCLUDING
LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE,
MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL
PROPERTY RIGHT. Intel products are not intended for use in medical, life saving, or life sustaining
applications.

Intel retains the right to make changes to its test specifications at any time, without notice.

The hardware vendor remains solely responsible for the design, sale and functionality of its product,
including any liability arising from product infringement or product warranty.

Performance tests and ratings are measured using specific computer systems and/or components and
reflect the approximate performance of Intel products as measured by those tests. Any difference in system
hardware or software design or configuration may affect actual performance. Buyers should consult other
sources of information to evaluate the performance of systems or components they are considering
purchasing. For more information on performance tests and on the performance of Intel products, call (U.S.)
1-800-628-8686 or 1-916-356-3104.

The Pentium® III Xeon™ processors, Pentium® 4 processors and Itanium® processors may contain design
defects or errors known as errata which may cause the product to deviate from published specifications.
Current characterized errata are available on request.

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

Copyright © Intel Corporation 2003

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

1 Overview ........................................................................................................................................ 4
  Motivation....................................................................................................................................... 4
  Prerequisites................................................................................................................................... 4
  Scope.............................................................................................................................................. 4
  Organization and Author Attribution.............................................................................................. 4
  Series Conventions.......................................................................................................................... 5

5 Memory Management .................................................................................................................. 6
  5.1 Avoiding Heap Contention Among Threads ............................................................................. 7
  5.2 Use Thread-Local Storage To Reduce Synchronization............................................................. 11
  5.3 Offset Thread Stacks To Avoid Cache Conflicts On Intel® Processors With
      Hyper-Threading Technology........................................................................................................ 16
1 Overview

Motivation

The objective of this series, which is comprised of this overview and four parts, is to provide guidelines for developing efficient multithreaded applications across Intel® architecture-based symmetric multiprocessors (SMP) and/or systems with Hyper-Threading Technology. An application developer can use the advice contained in this series to improve multithreading performance and to minimize unexpected performance variations on current as well as future SMP architectures built with Intel® processors.

This first version of this documentation provides general advice on multithreaded performance. Hardware-specific optimizations have deliberately been kept to a minimum. In future versions, topics covering hardware-specific optimizations will be added for developers who are willing to sacrifice portability for higher performance.

Prerequisites

Readers should have programming experience in a high-level language, preferably C, C++, and/or Fortran, though many of the recommendations in this document also apply to languages such as Java®, C#, and Perl. Readers must also understand basic concurrent programming and be familiar with one or more threading methods, preferably OpenMP®, POSIX threads (also referred to as Pthreads), or the Win32® threading API.

Scope

The main objective of these documents is to provide a quick reference to design and optimization guidelines for multithreaded applications on Intel® platforms. They are not intended to serve as a textbook on multithreading, nor do they represent a porting guide to Intel platforms.

Organization and Author Attribution

The “Developing Platform Consistent Threaded Applications” series covers topics ranging from general advice applicable to any multithreading method, to usage guidelines for Intel® software products, as well as API-specific issues. While designed as part of a series, each included chapter contains a discrete discussion of an important threading issue and can be read separately.
The included chapters and author attribution for each, are as follows:

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Scope</th>
<th>Contributing Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1, <strong>Overview</strong></td>
<td>This series overview</td>
<td>Bill Magro</td>
</tr>
<tr>
<td>Chapter 2, <strong>Intel® Software Development Products</strong></td>
<td>This chapter describes how to use Intel software products to develop, debug, and optimize multithreaded applications</td>
<td>Bruce Greer, Clay Breshears, Judi Goldstein, Martyn Corden, Phil Kerly, and Vasanth Tovinkere</td>
</tr>
<tr>
<td>Chapter 3, <strong>Application Threading</strong></td>
<td>This chapter covers general topics in parallel performance and occasionally refers to API-specific issues</td>
<td>Aaron Coday, Bill Magro, Clay Breshears, Henry Gabb, Prasad Kakulavarapu, Sanjiv Shah, and Vasanth Tovinkere</td>
</tr>
<tr>
<td>Chapter 4, <strong>Synchronization</strong></td>
<td>The sections in this chapter discuss techniques to mitigate the negative impact of synchronization on performance</td>
<td>Grant Haab, Henry Gabb, Prasad Kakulavarapu and Vasanth Tovinkere</td>
</tr>
<tr>
<td>Chapter 5, <strong>Memory Management</strong></td>
<td>Threads add another dimension to memory management that should not be ignored. This chapter covers memory issues that are unique to multithreaded applications</td>
<td>Clay Breshears, Jay Hoeflinger, Paul Petersen, and Phil Kerly</td>
</tr>
</tbody>
</table>

The user is free to download the entire series as a whole, or to download or read each chapter of interest as the need arises. Cross-references to related topics are provided throughout.

**Series Conventions**

‘This series’ refers to the above five chapters. Topics within chapters are referred to as ‘sections’. Cross-references are in outline notation, combining chapter and section numbers.
5 Memory Management

Adding concurrency to applications can improve performance in obvious ways. Other chapters in this series address many of the issues that can impact the performance of threaded applications. Avoiding contention for heap resources, using storage that is local to threads rather than shared to reduce synchronization, and carefully managing memory allocations are some of the less obvious, but no less important, considerations that can also impact threaded performance. These memory management issues are covered in this chapter.
5.1 Avoiding Heap Contention Among Threads

Category
Memory Management

Scope
General multithreading

Keywords
Heap contention, synchronization, dynamic memory allocation, lock contention, stack allocation

Abstract
Allocating memory from the system heap can be an expensive operation. To make allocation thread-safe, a lock is used to synchronize access to the heap. The contention on this lock can limit the performance benefits from multithreading. To solve this problem, change the allocation strategy to avoid using shared lock.

Background
The system heap (as used by malloc) is a shared resource. To make it safe to use by multiple threads it is necessary to add synchronization to gate access to the shared heap. Synchronization, in this case lock acquisition, requires two interactions (i.e., locking and unlocking, with the operating system – an expensive overhead.

The OpenMP* implementation in the Intel® 7.0 compilers exports two functions, kmp_malloc and kmp_free. These functions maintain a per-thread heap attached to each thread of the OpenMP team. Threads that call these functions avoid the use of the lock that protects access to the standard system heap. The threadprivate directive can be used as well to create a private copy of globally declared variables for each thread in the OpenMP team.

The Win32* HeapCreate function can be used to allocate separate heaps for all of the threads used by the application. The flag HEAP_NO_SERIALIZE is used to disable the use of synchronization on this new heap since only a single thread will access it.

If the heap handle is stored in a Thread Local Storage (TLS) location, this heap can be used whenever an application thread needs to allocate or free memory. Note that memory allocated in this manner must be explicitly released by the same thread that performs the allocation. For Pthreads applications, the pthread_key_create and pthread_{get|set}specific API can be used to obtain access to TLS but the management of this global storage is the programmer’s responsibility.

If you need to use a more general replacement (where the thread which allocates the memory is not necessarily the thread which releases the memory, then it may be more appropriate to look into using a commercial replacement to the heap manager as listed in the references section.
The following example uses several features of the Win32 API:

```c
#include <windows.h>

static DWORD tls_key;

__declspec (dllexport) void* thr_malloc (size_t n)
{
    return HeapAlloc (TlsGetValue (tls_key), 0, n);
}

__declspec (dllexport) void thr_free (void *ptr)
{
    HeapFree (TlsGetValue (tls_key), 0, ptr);
}

BOOL WINAPI DllMain (HINSTANCE hinstDLL,
    DWORD fdwReason,
    LPVOID lpReserved)
{
    switch (fdwReason)
    {
    case DLL_PROCESS_ATTACH:
        // Use Thread Local Storage to remember the heap
        tls_key = TlsAlloc ();
        TlsSetValue (tls_key, GetProcessHeap ());
        break;
    case DLL_THREAD_ATTACH:
        // Use HEAP_NO_SERIALIZE to avoid lock contention
        TlsSetValue
        {tls_key, HeapCreate (HEAP_NO_SERIALIZE, 0, 0));
        break;
    case DLL_THREAD_DETACH:
        HeapDestroy (TlsGetValue (tls_key));
        break;
    case DLL_PROCESS_DETACH:
        TlsFree (tls_key);
        break;
    }
    return TRUE;   // Successful DLL_PROCESS_ATTACH
}
```

First, it uses a dynamic load library (DLL) to allow the threads to be registered at the point of creation. It also uses TLS to remember the heap that is assigned to each thread. Finally, it uses the ability of the Win32 API to independently manage unsynchronized heaps.

**Advice**

In addition to the use of multiple independent heaps, it is also possible to incorporate other techniques to minimize the lock contention caused by a shared lock that is used to protect the system heap. If the memory is only to be accessed within a small lexical
context, the *alloca* routine can sometimes be used to allocate memory from the current stack frame. This memory is automatically deallocated upon function return.

A per-thread free list is another technique. Initially, memory is allocated from the system heap with *malloc*. When the memory would normally be released it is added to a per-thread linked-list. If the thread needs to reallocate memory of the same size, it can immediately retrieve the stored allocation from the list without going back to the system heap.

```c
struct MyObject
{
    struct MyObject *next;
};

static __declspec(thread) struct MyObject *freelist_MyObject = 0;

struct MyObject *malloc_MyObject ()
{
    struct MyObject *p = freelist_MyObject;
    if (p == 0)
        return malloc (sizeof (struct MyObject));
    freelist_MyObject = p->next;
    return p;
}

void free_MyObject (struct MyObject *p)
{
    p->next = freelist_MyObject;
    freelist_MyObject = p;
}
```

**Usage Guidelines**

With any optimization you encounter trade-offs. In this case the trade-off is in exchanging lower contention on the system heap for higher memory usage. When each thread is maintaining its own private heap or collection of objects, these areas are not available to other threads. This may result in a memory imbalance between the threads, similar to the load imbalance you encounter when threads are performing varying amount of work (see Application Threading, 3.3: Load Balance And Parallel Performance). The memory imbalance may cause the working set size to increase and the total memory usage by the application to also increase. The increase in memory usage usually has a minimal performance impact. An exception occurs when the increase in memory usage exhausts the available memory. If this happens it may cause the application to either abort or swap to disk.
References
In this series, see also:

Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTune™ Performance Analyzer
Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

See also:

MicroQuill SmartHeap for SMP
The HOARD memory allocator
Documentation for the following Win32 functions:

HeapAlloc, HeapCreate, HeapFree
TlsAlloc, TlsGetValue, TlsSetValue
Alloca
5.2 Use Thread-Local Storage To Reduce Synchronization

Category
Memory Management

Scope
General multithreading

Keywords
Thread-local storage, synchronization, OpenMP, Pthreads, Win32 threads

Abstract
Synchronization is often an expensive operation that can limit the performance of a multi-threaded program. Using thread-local data structures instead of data structures shared by the threads can reduce synchronization in certain cases, thus allowing a program to run faster.

Background
When data structures are shared by a group of threads and at least one thread is writing into them, synchronization between the threads is sometimes necessary to make sure that all threads see a consistent view of the shared data at all times. The typical synchronized access regime for threads in this situation is for a thread to acquire a lock, read or write the shared data structures, then release the lock.

All forms of locking have overhead to maintain the lock data structures and they use atomic instructions that slow down modern processors. Synchronization also slows down the program because it eliminates parallel execution inside the synchronized code, forming a serial execution bottleneck. Therefore, when synchronization occurs within a time-critical section of code, code performance can suffer.

The synchronization can be eliminated from the multithreaded, time-critical code sections if the program can be re-written to use thread-local storage instead of shared data structures. This is possible if the nature of the code is such that real-time ordering of the accesses to the shared data is unimportant. Synchronization can also be eliminated when the ordering of accesses is important, if the ordering can be safely postponed to execute during infrequent, non-time-critical sections of code.
Consider, for example, the use of a variable to count events that happen on several threads. The following code shows one way to write such a program in OpenMP:

```c
int count=0;
#pragma omp parallel shared(count)
{
    if (event_happened)
    {
        #pragma omp atomic
        count++;
    }
}
```

This program pays a price each time the event occurs because it must synchronize to guarantee that only one thread at a time increments `count`. Every event causes synchronization. Removing the synchronization makes the program run faster. One way to do this safely is to have each thread count its own events in the parallel region then sum the individual counts later. The following code demonstrates this technique:

```c
int count=0;
int tcount=0;
#pragma omp threadprivate(tcount)

#pragma omp parallel
{
    if (event_happened)
    {
        tcount++;
    }
}
#pragma omp parallel shared(count)
{
    #pragma omp atomic
    count += tcount;
}
```

This program uses a `tcount` variable that is private to each thread to store the count for each thread. After the first parallel region counts all the local events, a subsequent region adds this count into the overall count. This solution trades synchronization per event for synchronization per thread. Performance will improve if the number of events is much larger than the number of threads.

An additional advantage of using thread-local storage during time-critical portions of the program is that the data may stay live in a processor’s cache longer than shared data, if the processors do not share a data cache. When the same address exists in the data cache of several processors and is written by one of them, it must be invalidated in the caches of all other processors, causing it to be re-fetched from memory when the other processors access it. But thread-local data will never be written by any other processors and will therefore be more likely to remain in the cache of its processor.
Developing Platform Consistent Multithreaded Applications

The previous example code shows one way to specify thread-local storage in OpenMP. To do the same thing with Pthreads, the programmer must create a key to access thread-local storage, e.g.:

```c
#include <pthread.h>

pthread_key_t tsd_key;
<arbitrary data type> value;

if (pthread_key_create (&tsd_key, NULL))
    err_abort(status, "Error creating key");

if (pthread_setspecific( tsd_key, value))
    err_abort(status, "Error in pthread_setspecific");

value = (<arbitrary data type>)pthread_getspecific( tsd_key );
```

With the Win32 API, the programmer allocates a TLS index with `TlsAlloc` then uses that index to set a thread-local value, e.g.:

```c
DWORD tls_index;
LPVOID value;

tls_index = TlsAlloc();

if (tls_index == TLS_OUT_OF_INDEXES)
    err_abort( tls_index, "Error in TlsAlloc");

status = TlsSetValue( tls_index, value );

if (status == 0)
    err_abort( status, "Error in TlsSetValue");

value = TlsGetValue (tls_index);
```

In OpenMP, one can also create thread-local variables by specifying them in a `private` clause on the `parallel pragma` or the `threadprivate pragma`. These variables are automatically deallocated at the end of the parallel region. Of course, another way to specify thread-local data, regardless of the threading model, is to use variables allocated on the stack in a given scope. Such variables are deallocated at the end of the scope.

**Advice**

The technique of thread-local storage is applicable if synchronization is coded within a time-critical section of code, and if the operations being synchronized need not be ordered in real-time. If the real-time order of the operations is important, then the technique can still be applied if enough information can be captured during the time-critical section to reproduce the ordering later, during a non-time-critical section of code.
Consider the following example where threads write data into a shared buffer:

```c
int buffer[ENTRIES];

main()
{
    #pragma omp parallel
    {
        update_log (time, value1, value2);
    }
}

void update_log (time, value1, value2)
{
    #pragma omp critical
    {
        if (current_ptr + 3 > ENTRIES)
        {
            print_buffer_overflow_message ();
        }
        buffer[current_ptr] = time;
        buffer[current_ptr+1] = value1;
        buffer[current_ptr+2] = value2;
        current_ptr += 3;
    }
}
```

Let’s assume that `time` is some monotonically increasing value and the only real requirement of the program for this buffer data is that it be written to a file occasionally sorted according to `time`. We can eliminate the synchronization in the `update_log` routine by using thread-local buffers. Each thread allocates a separate copy of `tpbuffer` and `tpcurrent_ptr`. This allows us to eliminate the critical section in `update_log`. The entries from the various thread-private buffers can be merged later, in a non-time-critical portion of the program.

**Usage Guidelines**

One must be careful about the trade-offs involved in this technique. The technique does not remove the need for synchronization. It only moves the synchronization from a time-critical section of the code to a non-time-critical section of the code. First, determine whether the original section of code containing the synchronization is actually being slowed down significantly by the synchronization. (The Intel® VTune™ Performance Analyzer can be used to generate a performance profile.) Second, determine whether the time ordering of the operations is critical to the application. If not, synchronization can be removed, as in the event-counting code. If time ordering is critical, can the ordering be correctly re-constructed later? Third, verify that moving synchronization to another place in the code will not cause similar performance problems in the new location. One way to do this is to show that the number of synchronizations will decrease dramatically because of your work (such as in the event-counting example above).
Developing Platform Consistent Multithreaded Applications

References
In this series, see also:

Intel Software Development Products, 2.4: Find Multithreading Errors With The Intel Thread Checker

Intel Software Development Products, 2.5: Using Thread Profiler To Evaluate OpenMP Performance

Application Threading, 3.5: Expose Parallelism By Avoiding Or Removing Artificial Dependencies

See also:


5.3 Offset Thread Stacks To Avoid Cache Conflicts On Intel® Processors With Hyper-Threading Technology

Category
Memory Management

Scope
Multithreading with Pthreads or the Win32 API on Intel® processors with Hyper-Threading Technology

Keywords
Hyper-Threading Technology, cache-coherence, data alignment, VTune, stack allocation

Abstract
Hyper-Threading Technology-enabled processors share the first-level data cache on a cache-line basis among the logical processors. Frequent accesses to the virtual addresses on cache lines modulo 64 KB apart can cause alias conflicts that negatively impact performance. Since thread stacks are generally created on modulo 64 KB boundaries, accesses to the stack often conflict. By adjusting the start of the stack, the conflicts can be reduced and result in significant performance gains. Note that the 64 KB alias conflict is processor implementation dependent. Future processors may adjust the modulo boundary or eliminate this conflict altogether.

Background
Intel processors with Hyper-Threading Technology share the first-level data cache among logical processors. Cache lines whose virtual addresses are modulo 64 KB apart will conflict for the same slot in the first-level data cache. This can both affect the first-level data-cache performance and impact the branch prediction unit. In addition to 64 KB alias conflicts, it is possible to increase the number of branch mispredictions when the processor core logic uses speculative data with addresses modulo one megabyte apart. Under Microsoft Windows operating systems, thread stacks are currently created on a multiple of one megabyte boundaries by default. Two threads with very similar stack frame images and access patterns to local variables on the stack are very likely to cause alias conflicts resulting in substantial degradation. Future implementations of Intel processors with Hyper-Threading Technology will likely address both sources of alias conflicts. Adjusting the initial thread stack address of each thread is a simple workaround that can restore considerable performance to your application on Intel processors with Hyper-Threading Technology.

Advice
Create a stack offset for each thread to avoid first-level data cache-line conflicts between threads on Hyper-Threading-enabled processors.

There are two ways to determine if your application performance on Hyper-Threading enabled processors is suffering from these alias conflicts. The first, and most definitive, method is to try the suggested work-around across your application’s performance workloads. By comparing the resulting performance with and without Hyper-Threading technology enabled, you can directly measure the relative performance difference. The
second method is to use the Intel VTune Performance Analyzer. You will need to collect both clock tick events as well as 64 KB alias-conflict events across your application’s performance workloads with and without Hyper-Threading Technology enabled. After sorting the modules and functions in your application by clock ticks from highest to lowest, compare the number of 64 KB alias events. It’s not unusual to see an increase on the order of three times the number of 64 KB alias events with Hyper-Threading technology enabled. However, applications with a difference of eight times or greater at a module or function level have been shown to improve performance significantly using the optimization described below. If a sizeable portion of the total execution time is spent in the module or function, this will translate directly to an overall application level performance improvement.

Note that enabling or disabling Hyper-Threading support in Intel processors requires support in the system BIOS. Some BIOS implementations between vendors may not support user level access to enable or disable the Hyper-Threading feature.

Typically, threads are created using an operating system-specific application interface and passing it a pointer to a function as well as a pointer to a block of data specific to the thread. The key to adjusting the initial thread stack address is to replace the original function pointer with an intermediate function that can adjust the stack by a variable amount depending on the number of threads created. A new intermediate parameter block is needed that contains a pointer to the original thread function, a thread id, and a pointer to the original parameter data block. The intermediate function can adjust the stack address and then call the original function passing on the original thread specific parameter data. Using the new parameter block with a function pointer is a generic implementation that can be used for a pool of threads that may need to invoke different functions for a thread. As a less general alternative, you could avoid the function pointer technique and have the intermediate function call the original function directly. However, be careful that the compiler does not inline the original thread function within the alternative thread function. If the original thread function is ‘in-lined’, the benefit of the adjusted stack address for the original function is lost. Using the intermediate function method with a function pointer avoids this possibility because the compiler cannot determine which function to in-line at compile time.

The easiest way to adjust the initial stack address for each thread is to call the memory allocation function, _alloca, with varying byte amounts in the intermediate thread function. The _alloca function allocates memory directly on the stack. By adjusting the number of bytes passed to the _alloca function, you can adjust the next function’s starting stack address. The _alloca function is found in the malloc.h header file. Using this technique to adjust the stack address is allocating virtual memory in each thread’s stack frame that will go unused. In Example Code 1, a one kilobyte offset multiplied by the thread ID number is used to offset the thread stack frames. One kilobyte is not a magic number but one that has generally worked across various applications. One important point to note is that current versions of Microsoft Windows* operating systems have a limit on the amount of virtual memory accessible to a given process. If the limit on virtual memory is an important consideration for your application, you will need to determine the best offset or modify this technique within this constraint.
// Original thread parameter data structure
struct ParameterBlk
{
    int thread_specific_data;

    // Padding to keep thread data at least a cache-line apart
    char padding[2 * CACHE_LINE_SZ - sizeof (int)];
};

typedef DWORD (*PFI) (void*);

// Structure containing arguments provided to each thread
struct FunctionBlk
{
    PFI ThreadFuncPtr;
    struct ParameterBlk* function_parameters;
    unsigned int thread_number;

    // Padding to keep thread data at least a cache-line apart
    char padding[2 * CACHE_LINE_SZ - sizeof (PFI) -
                sizeof(struct ParameterBlk*) -
                sizeof(unsigned int)];
};

DWORD WINAPI OriginalThreadProc (LPVOID ptr)
{
    // This would have been the original thread function
    return 0;
}

#define STACK_OFFSET 1024

DWORD WINAPI IntermediateThreadProc (LPVOID ptr)
{
    struct FunctionBlk* parameter = (struct FunctionBlk*) ptr;

    // Adjusting stack address
    _alloca (parameter->thread_number * STACK_OFFSET);

    // Calling original thread procedure using a function pointer.

    // You could call the function directly as shown blow but be
    // careful that the function doesn’t get inlined.
    return (*parameter->ThreadFuncPtr)(parameter->function_parameters);
}

Example Code 1. Offsetting thread stacks with _alloca can avoid cache conflicts

When determining how many threads to create, you should consider using the main thread to do a portion of the work. The main thread is already likely to have a very different stack frame image and data-access pattern from the child threads that start with a clean stack frame aligned on one megabyte boundaries. Plus, there is one less child
thread to synchronize and manage. Note that this may not be desirable if the main thread must manage other tasks or be responsive to user input.

**Usage Guidelines**

A single-source implementation of the thread stack offsets can be used for multi-processor systems without performance impact. However, use of the stack offset can reduce the overall virtual memory available to an application. In general, this will affect only very large applications with a large number of threads. By adjusting the stack offset amount, you can balance performance needs versus virtual memory.

The best size for the stack offset is application-dependent. Thread functions that have deep thread stacks due to local variables with subsequent function calls or that operate on large local data structures within a loop tend to perform better with a larger stack-offset size. Conversely, thread functions with smaller stack sizes can perform well with a smaller stack offset. In general, increments of one kilobyte stack offsets per thread have worked well for many applications.

**References**

In this series, see also:

- Intel Software Development Products, 2.3: Avoiding And Identifying False Sharing Among Threads With The VTuneTM Performance Analyzer
- This chapter, 5.1: Avoiding Heap Contention Among Threads

See also: