Collections

The .NET Framework ships with a large number of collections, and it is not the purpose of this chapter to review every one of them — this is a task best left to the MSDN documentation. However, there are some non-trivial considerations that need to be taken into account when choosing a collection, especially for performance-sensitive code. It is these considerations that we will explore throughout this section.

Note Some developers are wary of using any collection classes but built-in arrays. Arrays are highly inflexible, non-resizable, and make it difficult to implement some operations efficiently, but they are known for having the most minimal overhead of any other collection implementation. You should not be afraid of using built-in collections as long as you are equipped with good measurement tools, such as those we considered in Chapter 2. The internal implementation details of the .NET collections, discussed in this section, will also facilitate good decisions. One example of trivia: iterating a `List<T>` in a `foreach` loop takes slightly longer than in a `for` loop, because `foreach`- enumeration must verify that the underlying collection hasn't been changed throughout the loop.

First, recall the collection classes shipped with .NET 4.5 — excluding the concurrent collections, which we discuss separately — and their runtime performance characteristics. Comparing the insertion, deletion, and lookup performance of collections is a reasonable way to arrive at the best candidate for your needs. The following table lists only the generic collections (the non-generic counterparts were retired in .NET 2.0):

There are several lessons to be learned from collection design, the choice of collections that made the cut to be included in the .NET Framework, and the implementation details of some collections:

- The storage requirements of different collections vary significantly. Later in this chapter we will see how internal collection layout affects cache performance with `List<T>` and `LinkedList<T>`. Another example is the `SortedSet<T>` and `List<T>` classes; the former is implemented in terms of a binary search tree with *n* nodes for *n* elements, and the latter in terms of a contiguous array of *n* elements. On a 32-bit system, storing *n* value types of size *s* in a sorted set requires (20 + *s*)*n* bytes of storage, whereas the list requires only *sn* bytes.
- Some collections have additional requirements of their elements to achieve satisfactory performance. For example, as we have seen in Chapter 3, any implementation of a hash table requires access to a good hash code for the hash table's elements.
- Amortized *O*(1) cost collections, when executed well, are hardly distinguishable from true *O*(1) cost collections. After all, few programmers (and programs!) are wary of the fact that `List<T>.Add` may sometimes incur a significant memory allocation cost and run for a duration of time that is linear in the number of list elements. Amortized time analysis is a useful tool; it has been used to prove optimality bounds on numerous algorithms and collections.
- The ubiquitous space-time tradeoff is present in collection design and even the choice of which collections to include in the .NET Framework. `SortedList<K,V>` offers very compact and sequential storage of elements at the expense of linear time insertion and deletion, whereas `SortedDictionary<K,V>` occupies more space and is non-sequential, but offers logarithmic bounds on all operations.

Note There's no better opportunity to mention strings, which are also a simple collection type — a collection of characters. Internally, the `System.String` class is implemented as an immutable, non-resizable array of characters. All operations on strings result in the creation of a new object. This is why creating long strings by concatenating thousands of smaller strings together is extremely inefficient. The `System.Text.StringBuilder` class addresses these problems as its implementation is similar to `List<T>`, and doubles its internal storage when it's mutated. Whenever you need to construct a string from a large (or unknown) number of smaller strings, use a `StringBuilder` for the intermediate operations.

This richness of collection classes may seem overwhelming, but there are cases when none of the built-in collections is a good fit. We will consider a few examples later. Until .NET 4.0, a common cause of complaint with respect to the built-in collections was the lack of thread safety: none of the collections in Table 5-1 is safe for concurrent access from multiple threads. In .NET 4.0 the `System.Collections.Concurrent` namespace introduces several new collections designed from the ground up for concurrent programming environments.

Table 5-1. *Collections in the .NET Framework*

Collection	Details	Insertion Time	Deletion Time	Lookup Time	Sorted	Index Access
`List<T>`	Automatically resizable array	Amortized $O(1)$*	$O(n)$	$O(n)$	No	Yes
`LinkedList<T>`	Doubly-linked list	$O(1)$	$O(1)$	$O(n)$	No	No
`Dictionary<K,V>`	Hash table	$O(1)$	$O(1)$	$O(1)$	No	No
`HashSet<T>`	Hash table	$O(1)$	$O(1)$	$O(1)$	No	No
`Queue<T>`	Automatically resizable cyclic array	Amortized $O(1)$	$O(1)$	--	No	No
`Stack<T>`	Automatically resizable array	Amortized $O(1)$	$O(1)$	--	No	No
`SortedDictionary<K,V>`	Red-black tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes (keys)	No
`SortedList<K,V>`	Sorted resizable array	$O(n)$**	$O(n)$	$O(\log n)$	Yes (keys)	Yes
`SortedSet<T>`	Red-black tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	Yes	No

** By "amortized" in this case we mean that there are operations that may take O(n) time, but most operations will take O(1) time, such that the average time across n operations is O(1).*

*** Unless the data are inserted in sort order, in which case O(1).*

Concurrent Collections

With the advent of the Task Parallel Library in .NET 4.0, the need for thread-safe collections became more acute. In Chapter 6 we will see several motivating examples for accessing a data source or an output buffer concurrently from multiple threads. For now, we focus on the available concurrent collections and their performance characteristics in the same spirit of the standard (non-thread-safe) collections we examined earlier.

Table 5-2.*Concurrent Collections in the .NET Framework*

Collection	Most Similar To	Details	Synchronization
`ConcurrentStack<T>`	`Stack<T>`	Singly linked list[1]	Lock-free (CAS), exponential backoff when spinning
`ConcurrentQueue<T>`	`Queue<T>`	Linked list of array segments (32 elements each)[2]	Lock-free (CAS), brief spinning when dequeueing an item that was just enqueued
`ConcurrentBag<T>`	--	Thread-local lists, work stealing[3]	Usually none for local lists, `Monitor` for work stealing
`ConcurrentDictionary<K,V>`	`Dictionary<K,V>`	Hash table: buckets and linked lists[4]	For updates: `Monitor` per group of hash table buckets (independent of other buckets) For reads: none

Notes on the "Details" column:

1. *In Chapter 6 we will see a sketch implementation of a lock-free stack using CAS, and discuss the CAS atomic primitive on its own merit.*
2. *The `ConcurrentQueue<T>` class manages a linked list of array segments, which allows it to emulate an unbounded queue with a bounded amount of memory. Enqueuing or dequeuing items involves only incrementing pointers into array segments. Synchronization is required in several locations, e.g., to ensure that items are not dequeued before the enqueuing thread has completed the enqueue operation. However, all synchronization is CAS-based.*
3. *The `ConcurrentBag<T>` class manages a list of items in no specific order. Items are stored in thread-local lists; adding or removing items to a thread-local list is done at the head of the list and usually requires no synchronization. When threads have to steal items from other threads' lists, they steal from the tail of the list, which causes contention only when there are fewer than three items in the list.*
4. *`ConcurrentDictionary<K,V>` uses a classic hash table implementation with chaining (linked lists hanging off each bucket; for a general description of a hash table's structure, see also Chapter 3). Locking is managed at the bucket level—all operations on a certain bucket require a lock, of which there is a limited amount determined by the constructor's `concurrencyLevel` parameter. Operations on buckets that are associated with different locks can proceed concurrently with no contention. Finally, all read operations do not require any locks, because all mutating operations are atomic (e.g., inserting a new item into the list hanging off its bucket).*

Although most concurrent collections are quite similar to their non-thread-safe counterparts, they have slightly different APIs that are affected by their concurrent nature. For example, the `ConcurrentDictionary<K,V>` class has helper methods that can greatly minimize the amount of locks required and address subtle race conditions that can arise when carelessly accessing the dictionary:

```
//This code is prone to a race condition between the ContainsKey and Add method calls:
Dictionary<string, int> expenses = ...;
if (!expenses.ContainsKey("ParisExpenses")) {
```

```
  expenses.Add("ParisExpenses", currentAmount);
} else {
  //This code is prone to a race condition if multiple threads execute it:
  expenses["ParisExpenses"] += currentAmount;
}
//The following code uses the AddOrUpdate method to ensure proper synchronization when
//adding an item or updating an existing item:
ConcurrentDictionary<string, int> expenses = ...;
expenses.AddOrUpdate("ParisExpenses", currentAmount, (key, amount) => amount + currentAmount);
```

The `AddOrUpdate` method guarantees the necessary synchronization around the composite "add or update" operation; there is a similar `GetOrAdd` helper method that can either retrieve an existing value or add it to the dictionary if it doesn't exist and then return it.

Cache Considerations

Choosing the right collection is not only about its performance considerations. The way data are laid out in memory is often more critical to CPU-bound applications than any other criterion, and collections affect this layout greatly. The main factor behind carefully examining data layout in memory is the CPU cache.

Modern systems ship with large main memories. Eight GB of memory is fairly standard on a mid-range workstation or a gaming laptop. Fast DDR3 SDRAM memory units are capable of ~ 15 ns memory access latencies, and theoretical transfer rates of ~ 15 GB/s. Fast processors, on the other hand, can issue billions of instructions per second; theoretically speaking, stalling for 15 ns while waiting for memory access can prevent the execution of dozens (and sometimes hundreds) of CPU instructions. Stalling for memory access is the phenomenon known as *hitting the memory wall*.

To distance applications from this wall, modern processors are equipped with several levels of *cache memory*, which has different internal characteristics than main memory and tends to be very expensive and relatively small. For example, one of the author's Intel i7-860 processor ships with three cache levels (see Figure 5-3):

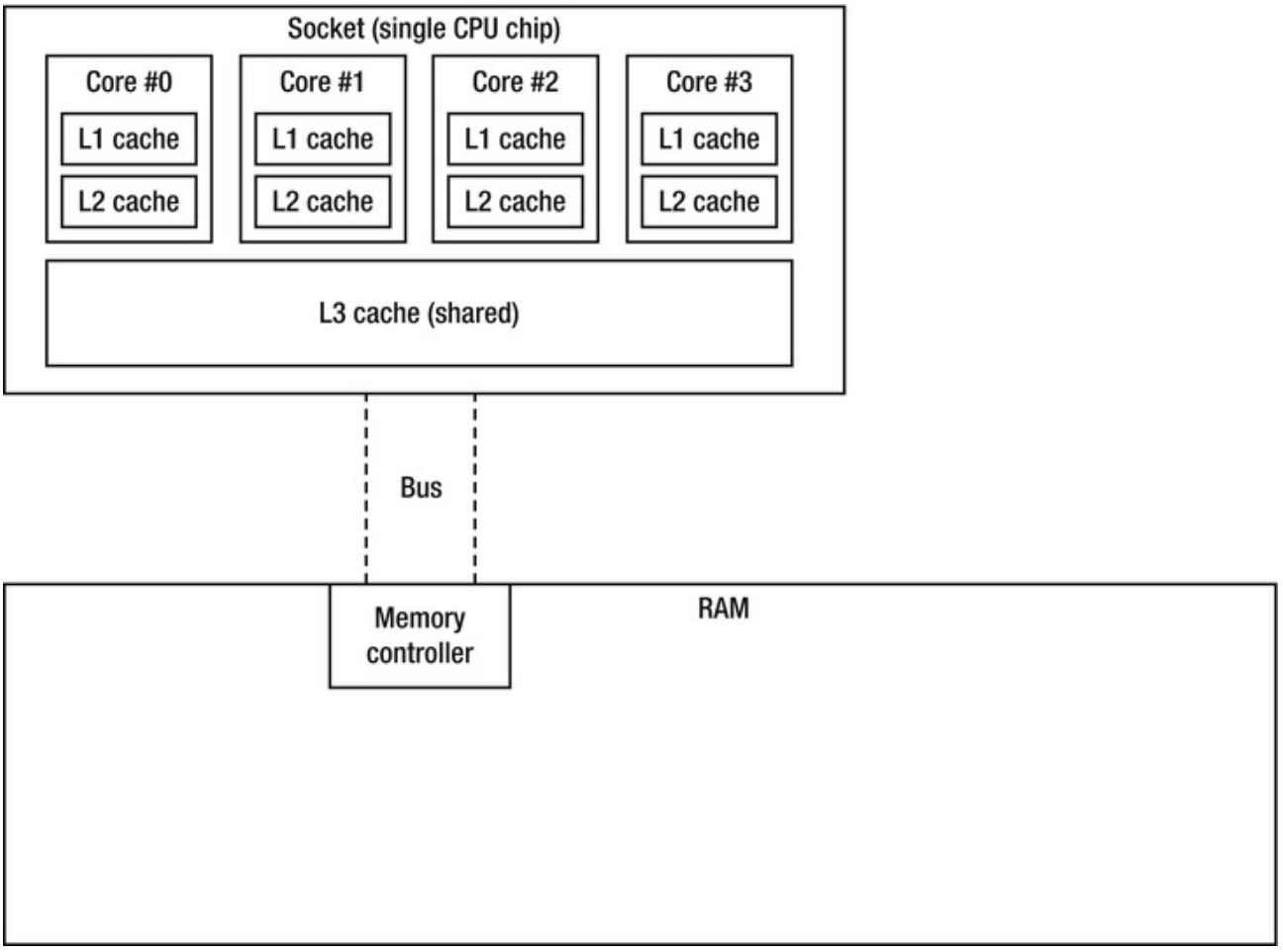

Figure 5-3. Intel i7-860 schematic of cache, core, and memory relationships

- Level 1 cache for program instructions, 32 KB, one for each core (total of 4 caches).
- Level 1 cache for data, 32 KB, one for each core (total of 4 caches).
- Level 2 cache for data, 256 KB, one for each core (total of 4 caches).
- Level 3 cache for data, 8 MB, shared (total of 1 cache).

When the processor attempts to access a memory address, it begins by checking whether the data is already in its L1 cache. If it is, the memory access is satisfied from the cache, which takes ~ 5 CPU cycles (this is called a *cache hit*). If it isn't, the L2 cache is checked; satisfying the access from the L2 cache takes ~ 10 cycles. Similarly, satisfying the access from L3 cache takes ~ 40 cycles. Finally, if the data isn't in any of the cache

levels, the processor will stall for the system's main memory (this is called a *cache miss*). When the processor accesses main memory, it reads from it not a single byte or word, but a *cache line*, which on modern systems consists of 32 or 64 bytes. Accessing any word on the same cache line will not involve another cache miss until the line is evicted from the cache.

Although this description does not do justice to the true hardware complexities of how SRAM caches and DRAM memories operate, it provides enough food for thought and discussion of how our high-level software algorithms can be affected by data layout in memory. We now consider a simple example that involves a single core's cache; in Chapter 6 we will see that multiprocessor programs can afflict additional performance loss by improperly utilizing the caches of multiple cores.

Suppose the task at hand is traversing a large collection of integers and performing some aggregation on them, such as finding their sum or average. Below are two alternatives; one accesses the data from a `LinkedList< int >` and the other from an array of integers (`int[]`), two of the built-in .NET collections.

```
LinkedList<int> numbers = new LinkedList<int>(Enumerable.Range(0, 20000000));
int sum = 0;
for (LinkedListNode<int> curr = numbers.First; curr != null; curr = curr.Next) {
  sum += curr.Value;
}
int[] numbers = Enumerable.Range(0, 20000000).ToArray();
int sum = 0;
for (int curr = 0; curr < numbers.Length; ++curr) {
  sum += numbers[curr];
}
```

The second version of the code runs *2× faster* than the first on the system mentioned above. This is a non-negligible difference, and if you only consider the number of CPU instructions issued, you might not be convinced that there should be a difference at all. After all, traversing a linked list involves moving from one node to the next, and traversing an array involves incrementing an index into the array. (In fact, without JIT optimizations, accessing an array element would require also a range check to make sure that the index is within the array's bounds.)

```
; x86 assembly for the first loop, assume 'sum' is in EAX and 'numbers' is in ECX
xor eax, eax
mov ecx, dword ptr [ecx+4] ; curr = numbers.First
test ecx, ecx
jz LOOP_END
LOOP_BEGIN:
add eax, dword ptr [ecx+10] ; sum += curr.Value
mov ecx, dword ptr [ecx+8] ; curr = curr.Next
test ecx, ecx
jnz LOOP_BEGIN ; total of 4 instructions per iteration

LOOP_END:
...
; x86 assembly for the second loop, assume 'sum' is in EAX and 'numbers' is in ECX
mov edi, dword ptr [ecx+4] ; numbers.Length
test edi, edi
jz LOOP_END
xor edx, edx ; loop index
LOOP_BEGIN:
add eax, dword ptr [ecx+edx*4+8] ; sum += numbers[i], no bounds check
inc edx
cmp esi, edx
jg LOOP_BEGIN ; total of 4 instructions per iteration

LOOP_END:
...
```

Given this code generation for both loops (and barring optimizations such as using SIMD instructions to traverse the array, which is contiguous in memory), it is hard to explain the significant performance difference by inspecting only the instructions executed. Indeed, we must analyze the memory access patterns of this code to reach any acceptable conclusions.

In both loops, each integer is accessed only once, and it would seem that cache considerations are not as critical because there is no reusable data to benefit from cache hits. Still, the way the data is laid out in memory affects greatly the performance of this program — not because the data is reused, but because of the way it is brought into memory. When accessing the array elements, a cache miss at the beginning of a cache line brings into the cache 16 consecutive integers (cache line = 64 bytes = 16 integers). Because array access is sequential, the next 15 integers are now in the cache and can be accessed without a cache miss. This is an almost-ideal scenario, with a 1:16 cache miss ratio. On the other hand, when accessing linked list elements, a cache miss at the beginning of a cache line can bring into cache *at most* 3 consecutive linked list nodes, a 1:4 cache miss ratio! (A node consists of a back pointer, forward pointer, and integer datum, which occupy 12 bytes on a 32-bit system; the reference type header brings the tally to 20 bytes per node.)

The much higher cache miss ratio is the reason for most of the performance difference between the two pieces of code we tested above. Furthermore, we are assuming the ideal scenario in which all linked list nodes are positioned sequentially in memory, which would be the case only if

they were allocated simultaneously with no other allocations taking place, and is fairly unlikely. Had the linked list nodes been distributed less ideally in memory, the cache miss ratio would have been even higher, and the performance even poorer.

A concluding example that shows another aspect of cache-related effects is the known algorithm for matrix multiplication by blocking. Matrix multiplication (which we revisit again in Chapter 6 when discussing C++ AMP) is a fairly simple algorithm that can benefit greatly from CPU caches because elements are reused several times. Below is the naïve algorithm implementation:

```
public static int[,] MultiplyNaive(int[,] A, int[,] B) {
  int[,] C = new int[N, N];
  for (int i = 0; i < N; ++i)
   for (int j = 0; j < N; ++j)
   for (int k = 0; k < N; ++k)
   C[i, j] += A[i, k] * B[k, j];
  return C;
}
```

In the heart of the inner loop there is a scalar product of the *i*-th row in the first matrix with the *j*-th column of the second matrix; the entire *i*-th row and *j*-th column are traversed. The potential for reuse by caching stems from the fact that the entire *i*-th row in the output matrix is calculated by repeatedly traversing the *i*-th row of the first matrix. The same elements are reused many times. The first matrix is traversed in a very cache-friendly fashion: its first row is iterated completely *N* times, then its second row is iterated completely *N* times, and so on. This does not help, though, because after the outer loop is done with iteration *i*, the method it does not the *i*-th row again. Unfortunately, the second matrix is iterated in a very cache-unfriendly fashion: its first *column* is iterated completely *N* times, then its second *column*, and so on. (The reason this is cache-unfriendly is that the matrix, an `int[,]`, is stored in memory in row-major order, as Figure 5-4 illustrates.)

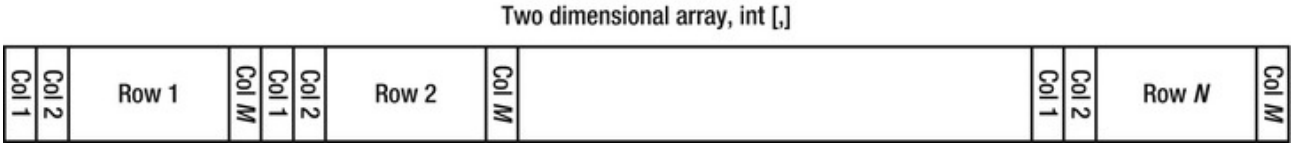

Figure 5-4. The memory layout of a two dimensional array (int[,]). Rows are consecutive in memory, columns are not

If the cache was big enough to fit the entire second matrix, then after a single iteration of the outer loop the entire second matrix would be in the cache, and subsequent accesses to it in column-major order would still be satisfied from cache. However, if the second matrix does not fit in the cache, cache misses will occur very frequently: a cache miss for the element *(i,j)* would produce a cache line that contains elements from row *i* but no additional elements from column *j*, which means a cache miss for every access!

Matrix multiplication by blocking introduces the following idea. Multiplying two matrices can be performed by the naïve algorithm above, or by splitting them into smaller matrices (blocks), multiplying the blocks together, and then performing some additional arithmetic on the results.

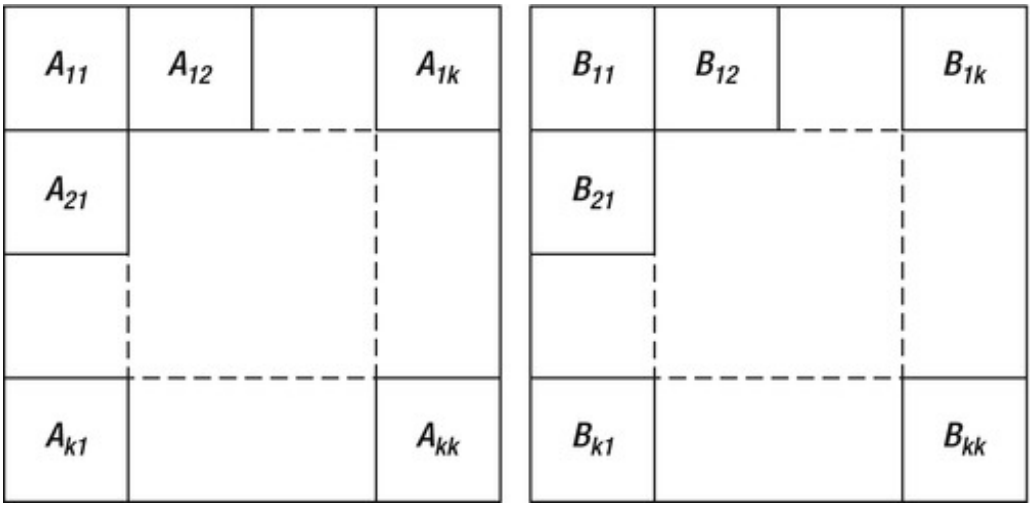

Figure 5-5. Matrices A and B given in block form, k × k blocks each

Specifically, if the matrices *A* and *B* are given in block form, as in Figure 5-5, then the matrix $C = AB$ can be calculated in blocks, such that $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j} + \ldots + A_{ik}B_{kj}$. In practice, this leads to the following code:

```
public static int[,] MultiplyBlocked(int[,] A, int[,] B, int bs) {
  int[,] C = new int[N, N];
  for (int ii = 0; ii < N; ii += bs)
   for (int jj = 0; jj < N; jj += bs)
     for (int kk = 0; kk < N; kk += bs)
      for (int i = ii; i < ii + bs; ++i)
        for (int j = jj; j < jj + bs; ++j)
        for (int k = kk; k < kk + bs; ++k)
        C[i, j] += A[i, k] * B[k, j];
```

```
    return C;
}
```

The apparently complex *six* nested loops are quite simple—the three innermost loops perform naïve matrix multiplication of two blocks, the three outermost loops iterate over the blocks. To test the blocking multiplication algorithm, we used the same machine from the previous examples (which has an 8 MB L3 cache) and multiplied 2048 × 2048 matrices of integers. The total size of both matrices is 2048 × 2048 × 4 × 2 = 32 MB, which does not fit in the cache. The results for different block sizes are shown in Table 5-3. In Table 5-3 you can see that blocking helps considerably, and that finding the best block size can have a significant secondary effect on performance:

Table 5-3. *Timing results of blocking multiplication for varying block sizes*

	Naïve (no blocks)	bs=4	bs=8	bs=16	bs=32	bs=64	bs=512	bs=1024
Time (s)	178	92	81	81	79	106	117	147

There are many additional examples where cache considerations are paramount, even outside the arena of algorithm design and collection optimization. There are also some even more refined aspects of cache and memory-related considerations: the relationships between caches at different levels, effects of cache associativity, memory access dependencies and ordering, and many others. For more examples, consider reading the concise article by Igor Ostrovsky, "Gallery of Processor Cache Effects" (http://igoro.com/archive/gallery-of-processor-cache-effects/, 2010).