CHAPTER 9

Cache Interference

in which we see how cache interference can ruin a parallel program's performance; we study how to design programs to eliminate cache interference; and we measure the effect on our program's running time

CHAPTER 9 Cache Interference

9.1 Origin of Cache Interference

To understand why the SMP parallel program for AES key search from Chapter 7 performs as poorly as the metrics in Chapter 8 show, we have to take a detour into the internals of the JVM and the CPU.

Here again are the per-thread variables used by each thread in the FindKeySmp program's parallel team. These are declared as instance fields of the parallel for loop subclass and are initialized in the parallel for loop's `start()` method.

```
new ParallelTeam().execute (new ParallelRegion()
   {
   public void run() throws Exception
      {
      execute (0, maxcounter, new IntegerForLoop()
         {
         // Thread local variables.
         byte[] trialkey;
         byte[] trialciphertext;
         AES256Cipher cipher;

         // Set up thread local variables.
         public void start()
            {
            trialkey = new byte [32];
            System.arraycopy
               (partialkey, 0, trialkey, 0, 32);
                trialciphertext = new byte [16];
                cipher = new AES256Cipher (trialkey);
                }
```

When one of the parallel team threads creates an instance of the parallel for loop subclass, the JVM allocates a block of storage in the computer's main memory to hold the parallel for loop object's instance fields. The storage block consists of four bytes to store the reference to the `trialkey` array, four bytes to store the reference to the `trialciphertext` array, four bytes to store the reference to the cipher object, and a few extra bytes of overhead for the JVM's internal use. (This assumes a 32-bit CPU where memory addresses are 32 bits, or 4 bytes.)

When the `trialkey` variable is initialized in the parallel for loop's `start()` method with the expression `new byte [32]`, the JVM allocates another block of storage to hold the byte array and stores a reference to this block in the `trialkey` variable. The block consists of JVM overhead plus 32 bytes for the array elements. Likewise, the `trialciphertext` and `cipher` variables each get a block of storage. The `cipher` object is an instance of class AES256Cipher, so storage blocks are also allocated for the object itself and for its instance fields. When all the threads have finished initializing their per-thread variables, the storage blocks might end up arranged in main memory somewhat as shown in Figure 9.1. (Actually, the JVM is allowed to place each storage block wherever it wants; the JVM is not required to allocate one thread's storage blocks contiguously.)

However, the computer's CPUs do not access the main memory directly. As described in Chapter 2, a **cache memory** sits between the CPU and the main memory. When the CPU needs to read a memory location, it first reads the appropriate **cache line**, a block of contiguous memory locations that includes the desired location, from main memory into the cache. The CPU then reads the contents of the memory location from the cache. Subsequent reads of the same memory location, or any location in the same cache line, come from the fast cache rather than the slow main memory, thus speeding up the program's execution. The cache line size depends on the CPU hardware; it is typically 64 bytes or 128 bytes.

Figure 9.2 shows the cache line boundaries superimposed on the program variables' storage blocks. Note that while the cache line boundaries occur regularly every 64 or 128 memory locations, the boundaries between the variables' storage blocks can fall at any locations, depending on the sizes of the blocks. Therefore, it is quite possible for a particular cache line to contain pieces of different variables' storage blocks. In particular, it is possible for a particular cache line to contain pieces of *different threads' per-thread variables'* storage blocks—as is the case for the cache line in the middle of Figure 9.2.

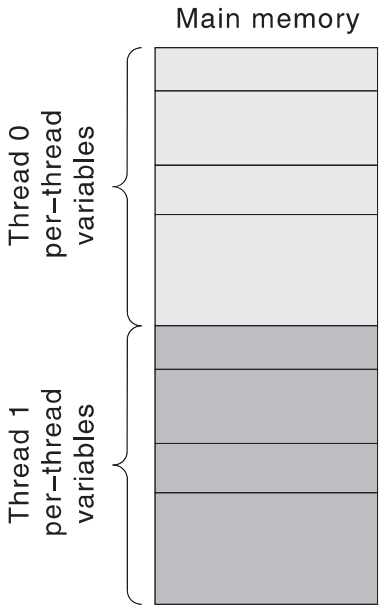

Figure 9.1 Per-thread variable memory layout

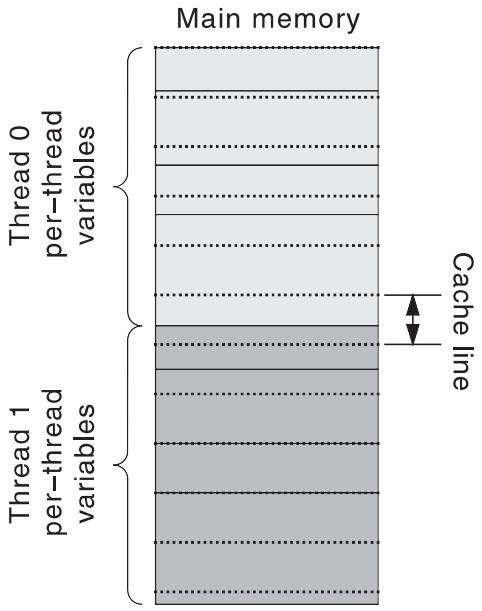

Figure 9.2 Memory layout showing cache line boundaries

Recall from Chapter 2 that one example of a cache-management strategy involves a *write-through* policy for dirty cache lines and an *invalidation*-based cache-coherence protocol. Suppose one CPU has

read a certain variable, so the variable's cache line has been loaded into that CPU's cache, and then that CPU writes a value into the variable. The cache's contents no longer agree with the main memory's contents, so the cache line is written from the cache back out to main memory. Furthermore, the writing CPU sends a signal telling the other CPUs to invalidate the cache line in question in the other CPUs' caches. This causes the other CPUs to read the cache line's new contents from the main memory the next time the other CPUs access the cache line. This is how a thread executing in one CPU obtains a new value stored into a shared variable by a thread executing in another CPU.

Now, consider what happens when one thread reads its own *per-thread* variable located in a certain cache line and another thread reads its own *per-thread* variable located in the same cache line. Each CPU loads the same cache line into its own cache, and then each CPU proceeds to read its own per-thread variable from the cache line. Suppose the first thread writes a new value into its own per-thread variable. The first thread's write causes the second thread's cache line to become invalidated. Suppose the second thread reads its own per-thread variable a second time. Normally, this variable would be read from the fast cache without needing to go to the main memory. But because the cache line was invalidated, the second thread's CPU has to reload the cache line from slow main memory before it can get the variable's value—even though the second thread's variable's value did not change. Thus, the overall program takes longer to run than it would if the threads never wrote the same cache lines. This effect is called **cache interference**.

Even though the threads never read or write the same *memory locations* (because each thread its accessing only its own per-thread variables), the threads do read and write the same *cache line* (because the JVM put different threads' per-thread variables at addresses that fell in the same cache line). The resulting performance reduction is similar to what happens if the threads have to take turns reading and writing a *shared* variable. But because the threads are not actually sharing the variables, the phenomenon is called **false sharing**.

Cache interference, then, is the reason the AES key search program fails to achieve the expected performance. The more threads (processors) there are, the more opportunities there are for each CPU to invalidate the other CPU's cache lines, and the greater the detriment to the program's performance.

9.2 Eliminating Cache Interference

To eliminate the cache interference that arises from false sharing of per-thread variables, we must ensure that different threads' per-thread variables never reside in the same cache line. One way to accomplish this would be for the JVM to allocate each storage block starting at a cache line boundary. However, JVM implementations don't do this because of all the wasted space that would result in the cache lines at the ends of the allocated blocks. A better way to avoid cache interference would be for the JVM to allocate the storage blocks for each thread's per-thread variables in separate regions of memory, with no overlapping cache lines. However, the Java language has no way for the programmer to declare variables as per-thread variables for purposes of memory allocation. (While the Java platform does provide class java.lang.ThreadLocal, this class does not cause variables to be allocated in different cache lines.)

Another way to eliminate cache interference is to add some **padding** in the memory layout. Suppose every per-thread variable's storage block has some extra bytes at the end, enough bytes to spill across the next cache line boundary. Suppose the program never reads or writes these extra padding bytes. Figure 9.3 shows the result. In Figure 9.3, the accessed portion of each variable's storage block is

shaded, and the padding (non-accessed) portion is white. You can see that the shaded portion of one variable is never in the same cache line as the shaded portion of another variable. Consequently, cache interference does not occur.

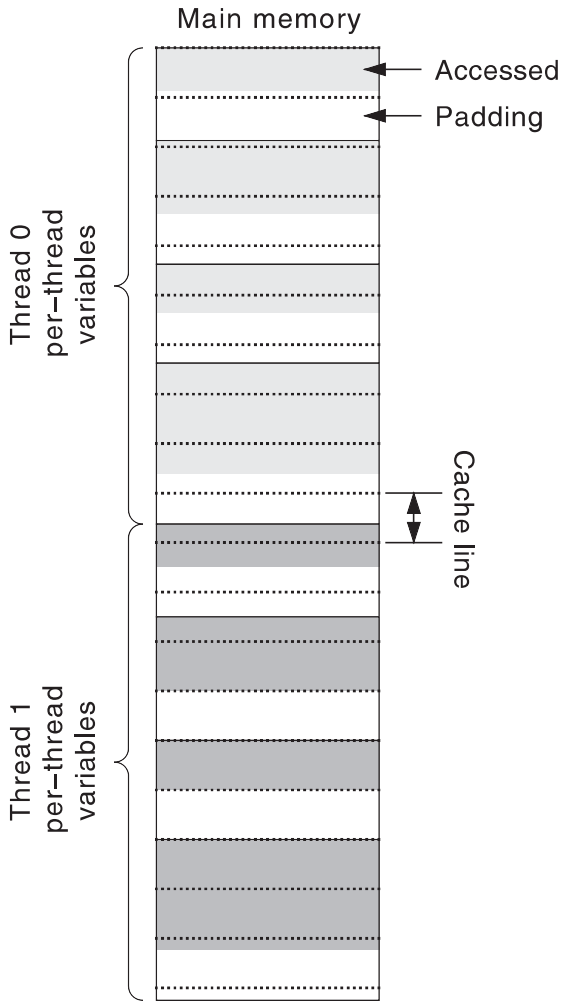

Figure 9.3 Memory layout with extra padding

Here's how to get that extra padding. First, when declaring a class's instance fields, throw in some extra padding fields. Not knowing what the cache line size will be when we run the program, we'll be conservative and add 128 bytes of padding. Sixteen fields of type `long`, each of which occupies eight bytes, does the job. Here are the AES key search program's parallel for loop subclass's instance fields, with padding.

```
 // Thread local variables.
byte[] trialkey;
byte[] trialciphertext;
AES256Cipher cipher;
```

```
// Extra padding.
long p0, p1, p2, p3, p4, p5, p6, p7;
long p8, p9, pa, pb, pc, pd, pe, pf;
```

Note that getting the padding by declaring a 16-element `long` array instance field (or a 128-element `byte` array instance field) *does not work.* The problem is that only a four-byte *reference* to the array ends up in the parallel for loop subclass's storage block; the actual array elements end up in a separate block. The padding fields must be of a primitive type, such as `long`, for them to be located in the parallel for loop subclass's storage block.

Second, when creating a new array, allocate enough extra array elements to occupy 128 bytes. Here is the parallel for loop subclass's `start()` method, allocating extra padding in the byte arrays.

```
// Set up thread local variables.
public void start()
   {
   trialkey = new byte [32+128]; // + padding
   System.arraycopy
      (partialkey, 0, trialkey, 0, 32);
   trialciphertext = new byte [16+128]; // + padding
   cipher = new AES256CipherSmp (trialkey);
   }
```

The `start()` method also creates an instance of class AES256CipherSmp instead of class AES256Cipher. Class AES256CipherSmp has extra padding in *its* instance fields. To eliminate cache interference, *all* the per-thread storage blocks must include the extra padding.

This technique of adding padding bytes is not a perfect solution to the problem of cache interference. It increases the amount of storage the program consumes. It requires either knowing the machine's cache line size (which reduces the program's portability) or picking an amount of padding one hopes is larger than any machine's cache line size (which may increase storage usage unnecessarily). But most of all, it requires that the programmer add code to do something the Java compiler or JVM should do. Perhaps a future version of the Java platform will automatically place per-thread variables in memory so as to avoid false sharing.

9.3 FindKeySmp3 Measurements

Program FindKeySmp3 in the Parallel Java Library is the same as program FindKeySmp from Chapter 7, except it incorporates padding to avert cache interference, as described earlier. To see whether the padding made a difference, we measure the FindKeySmp3 program's running time on the same "parasite" SMP parallel computer with the same input data sets as the FindKeySmp program. As it happens, the "parasite" computer's cache line size is 128 bytes.

Table 9.1 (at the end of the chapter) gives the running-time measurements in milliseconds for the modified AES key search program for various problem sizes *N*, as well as the speedups, efficiencies, and *EDSFs* calculated from the running times. Figure 9.4 plots the running-time metrics versus the number of processors. The *T* versus *K* plot looks a lot better than the FindKeySmp program's plot—it has nice straight lines. The *Speedup* versus *K* and *Eff* versus *K* plots also look dramatically better, just as they should according to Amdahl's Law, with efficiencies greater than 0.92 for all problem sizes. Adding extra padding in the memory layout has definitely eliminated the effects of cache interference.

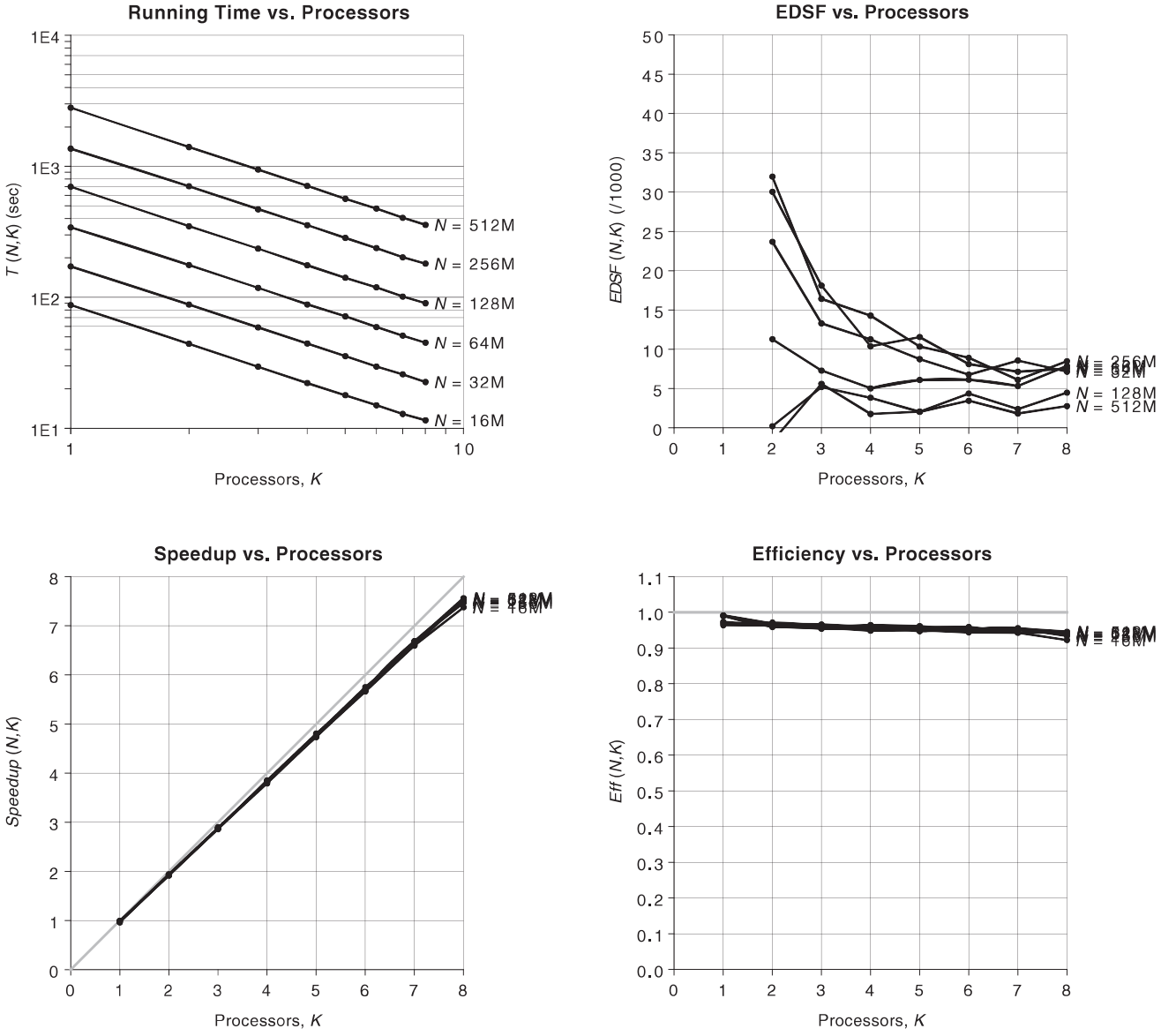

Figure 9.4 FindKeySeq/FindKeySmp3 running-time metrics

The *EDSF* versus *K* plot still looks strange; the curves are not even close to being horizontal lines. The fluctuations we see are due to two things: random measurement error, and the JVM's JIT compiler.

To illustrate just how the JIT compiler affects a Parallel Java program's performance, Table 9.2 (at the end of the chapter) lists the running-time metrics for a problem size of $N = 16M$ with the JIT compiler disabled. Figure 9.5 compares the running-time metrics for $N = 16M$ with and without the JIT compiler. From the plots, it's apparent that the JIT compiler introduces a small but definite reduction in the speedup and efficiency. Without the JIT compiler, the *EDSF* plot is nearly constant, as it should be, with small fluctuations due to random measurement error. With the JIT compiler enabled, the sequential fraction increases, as does the variation in the *EDSF* curve. The JIT compiler reduces the speedup, reduces the efficiency, and increases the sequential fraction by taking a bit of CPU time away from the program's computations to compile the hot spots to machine code.

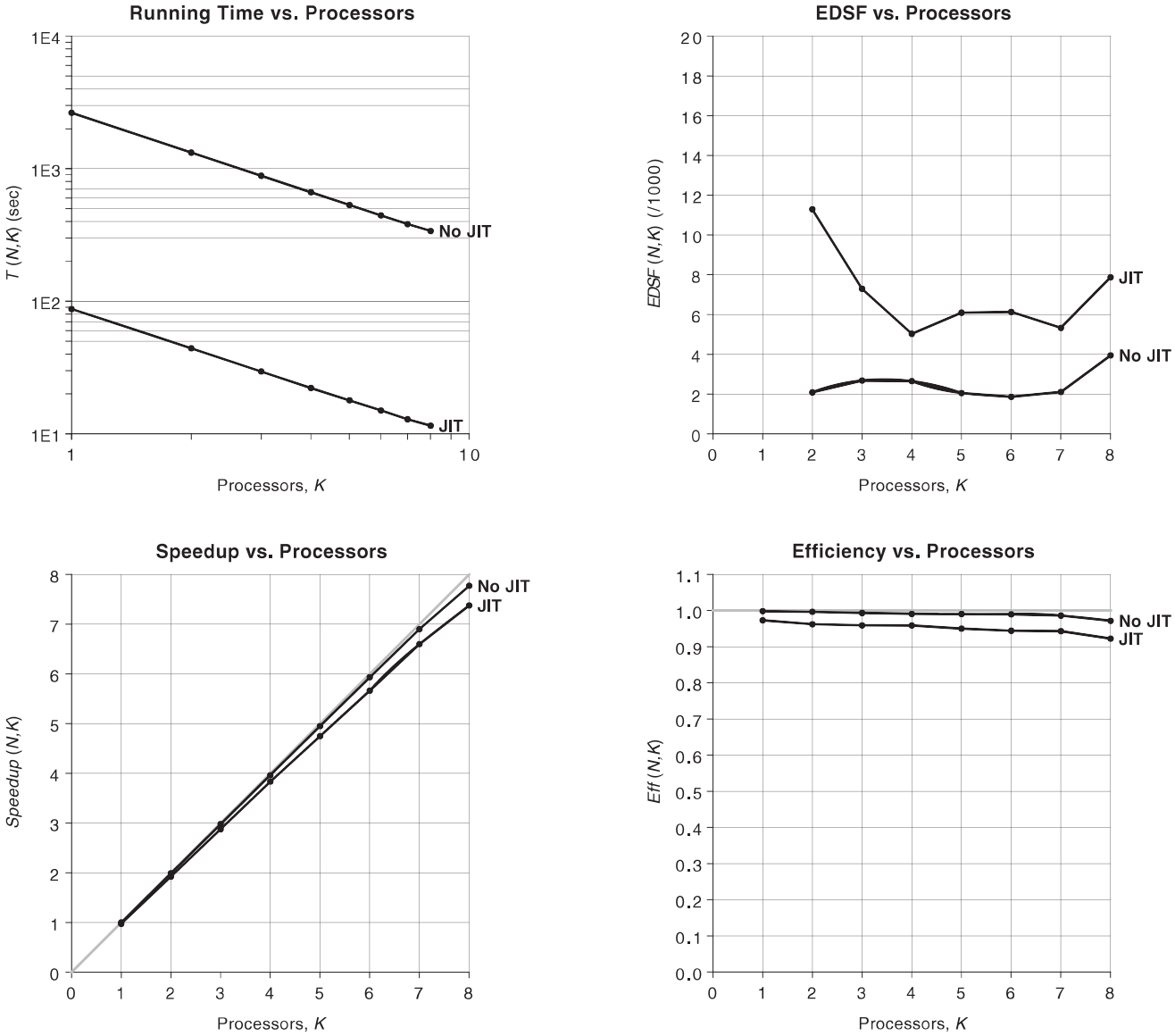

Figure 9.5 FindKeySeq/FindKeySmp3 running time metrics for $N = 16M$, with and without the JIT compiler

If the JIT compiler has negative effects on the speedup, efficiency, and sequential fraction, why not run with the JIT compiler disabled all the time? The plot of running time versus processors makes the answer clear. Without the JIT compiler, the program's running time increases by a factor of 30. The CPU

can execute native machine code instructions 30 times faster than it can interpret Java bytecodes. A slight impact on the speedup, efficiency, and *EDSF* is well worth such a drastic reduction in the running time. We always run our Parallel Java programs with the JIT compiler enabled (which is the default).

Now that we've dealt with the cache interference in the AES key search program, in Chapter 10 we'll return to Gustafson's observation and look at what happens when we scale up the problem size as we scale up the number of parallel processors.

9.4 For Further Information

On cache-related and other issues affecting Java program performance on SMP parallel computers:

- Z. Cao, W. Huang, and J. Chang. A study of Java virtual machine scalability issues on SMP systems. In *Proceedings of the 2005 IEEE International Workload Characterization Symposium*, 2005, pages 119–128.

\

Table 9.1 FindKeySeq/FindKeySmp3 running-time metrics

N	*K*	*T*	*Spdup*	*Eff*	*EDSF*	*N*	*K*	*T*	*Spdup*	*Eff*	*EDSF*
16M	seq	84981				128M	seq	677725			
16M	1	87346	0.973	0.973		128M	1	699172	0.969	0.969	
16M	2	44166	1.924	0.962	0.011	128M	2	348984	1.942	0.971	-0.002
16M	3	29540	2.877	0.959	0.007	128M	3	235672	2.876	0.959	0.006
16M	4	22166	3.834	0.958	0.005	128M	4	175724	3.857	0.964	0.002
16M	5	17895	4.749	0.950	0.006	128M	5	140989	4.807	0.961	0.002
16M	6	15004	5.664	0.944	0.006	128M	6	119072	5.692	0.949	0.004
16M	7	12877	6.599	0.943	0.005	128M	7	101317	6.689	0.956	0.002
16M	8	11520	7.377	0.922	0.008	128M	8	90134	7.519	0.940	0.004
32M	seq	170452				256M	seq	1350158			
32M	1	171926	0.991	0.991		256M	1	1364950	0.989	0.989	
32M	2	87999	1.937	0.968	0.024	256M	2	704288	1.917	0.959	0.032
32M	3	58833	2.897	0.966	0.013	256M	3	469923	2.873	0.958	0.016
32M	4	44433	3.836	0.959	0.011	256M	4	355856	3.794	0.949	0.014
32M	5	35588	4.790	0.958	0.009	256M	5	284293	4.749	0.950	0.010
32M	6	29624	5.754	0.959	0.007	256M	6	237631	5.682	0.947	0.009
32M	7	25824	6.601	0.943	0.009	256M	7	202138	6.679	0.954	0.006
32M	8	22568	7.553	0.944	0.007	256M	8	180758	7.469	0.934	0.008
64M	seq	339088				512M	seq	2704213			
64M	1	342194	0.991	0.991		512M	1	2805340	0.964	0.964	
64M	2	176237	1.924	0.962	0.030	512M	2	1402986	1.927	0.964	0.000
64M	3	118196	2.869	0.956	0.018	512M	3	944822	2.862	0.954	0.005
64M	4	88214	3.844	0.961	0.010	512M	4	709384	3.812	0.953	0.004
64M	5	71601	4.736	0.947	0.012	512M	5	565657	4.781	0.956	0.002
64M	6	59347	5.714	0.952	0.008	512M	6	475636	5.685	0.948	0.003
64M	7	50980	6.651	0.950	0.007	512M	7	405143	6.675	0.954	0.002
64M	8	45032	7.530	0.941	0.008	512M	8	357483	7.565	0.946	0.003

Table 9.2 FindKeySeq/FindKeySmp3 running time-metrics without JIT compiler

N	*K*	*T*	*Spdup*	*Eff*	*EDSF*
16M	seq	2634676			
16M	1	2638909	0.998	0.998	
16M	2	1322203	1.993	0.996	0.002
16M	3	884351	2.979	0.993	0.003
16M	4	664967	3.962	0.991	0.003
16M	5	532116	4.951	0.990	0.002
16M	6	443904	5.935	0.989	0.002
16M	7	381755	6.901	0.986	0.002
16M	8	338981	7.772	0.972	0.004