Wait-Free Shared-Memory Irradiance Caching

Kurt Debattista and Piotr Dubla ▪ *University of Warwick*

Luís Paulo Peixoto dos Santos ▪ *Universidade do Minho*

Alan Chalmers ▪ *University of Warwick*

Parallelizing rendering algorithms to exploit multiprocessor and multicore machines isn't straightforward. For example, the irradiance cache (IC) is an acceleration data structure that caches indirect diffuse irradiance values. In multicore systems, threads must share the IC to achieve high efficiency. A novel wait-free access mechanism significantly reduces synchronization overhead.

Desktop multicore computing calls for modifying traditional rendering algorithms to parallelize the available resources and maximize their use. For certain algorithms, such as classic ray tracing, this conversion can be relatively straightforward, but computing more complex lighting conditions requires careful consideration of thread synchronization to minimize overhead and allow computation.[1]

We've developed a parallel solution to one method used for computing complex lighting conditions—namely, the irradiance cache (IC), an acceleration data structure that caches sparsely sampled values for indirect diffuse irradiance in the framework of a distributed ray-tracing algorithm.[2] However, because all rendering threads can write to and read from the IC, a multithreaded shared-memory system must have a data-access-control mechanism to ensure that the data structure isn't corrupted.

Such control mechanisms incur their own overhead, so they must be carefully designed to not compromise performance. Traditionally, data-access-control mechanisms use lock-based mutual exclusion. However, nonblocking data structures that take the form of obstruction-free, lock-free, or wait-free data structures offer considerable performance advantages. Wait-free structures are the most powerful (see the "Nonblocking Synchronization" sidebar), but system developers and researchers have considered them difficult to construct, and they're relatively rare in practice.[3]

Here, we present an efficient wait-free algorithm that lets all threads concurrently access an unbounded shared IC, without using locks or critical sections. This is an extension of our previous research,[4] which presented an initial version of the wait-free algorithm and tested it on an eight-core machine. The earlier algorithm could handle only fixed-size arrays in the IC. Furthermore, it discarded some data when a conflict among threads occurred. We've fixed these limitations to guarantee the successful insertion of all new irradiance values. We've also assessed this algorithm's efficiency on two highly concurrent multicore systems with up to 24 physical cores. The results demonstrate its superior performance and scalability over two other more traditional and straightforward mechanisms for sharing the IC among a shared-memory system's threads. (An audiovisual presentation that augments this article's description of the algorithm is available at http://doi.ieeecomputersociety.org/10.1109/MCG.2010.80.)

Related Work in Parallel IC

Recent ray-tracing improvements have enabled interactive computation of many global effects, such as specular phenomena and correct shadows.[5] However, the dense sampling at each shading point required by indirect diffuse interreflections dramatically increases rendering times. Gregory Ward and his colleagues exploited a continuous smooth

Nonblocking Synchronization

Using shared-memory multithreading to execute parallel computations requires careful design of the access to data structures that all threads can access concurrently. A data-access-control mechanism must be able to ensure that the data structure isn't corrupted.

Traditionally, access control to shared-memory data structures is maintained via mutual exclusion. When critical sections are reasonably large, threads can use blocking mechanisms to preempt the running thread. However, when frequent access to a shared data structure is required, blocking costs can be prohibitive. In such cases, a thread would typically enter a busy-wait state, usually using a spin lock, when another thread is in the critical section and maintain that state until the other thread completes. Such control mechanisms incur overheads, such as serialization of accesses to the shared data structure. Blocking entails expensive context switches, and busy-waiting of frequently accessed resources leads to contention that can drastically reduce performance as the number of threads increases.[1]

Alternatives that avoid mutual exclusion exist in the form of nonblocking synchronization. By carefully ordering instructions, a system developer can remove all critical sections and thereby reduce contention by allowing nonblocking algorithms to guarantee that none of the code is serialized.[2]

The weakest form of nonblocking data structures is *obstruction-free methods*, which guarantee that a thread can complete in finite time if it operates in isolation. When nonblocking data structures can guarantee that at least one among a set of concurrent threads will complete in finite time, they're said to be *lock free*. All lock-free algorithms are obstruction free. However, lock-free and obstruction-free methods rely on retrials and can't guarantee an upper bound on the number of executed instructions.

When an algorithm can guarantee that all threads will complete in finite time, the algorithm is said to be *wait free.* Wait-free algorithms can guarantee an upper bound on the number of instructions, thus avoiding starvation, deadlock, livelock, and priority-inversion problems. Wait-free algorithms are ideal for multiprogrammed multiprocessors—for example, when a thread holding a lock is preempted and causes all other threads to busy-wait uselessly. Clearly, all wait-free data structures are also lock free.

The construction of nonblocking algorithms requires powerful atomic primitives, which execute without interruption as a single instruction on modern architectures. We can view these algorithms as a limiting case that reduces the size of critical sections to the size of individual machine instructions.

```
atomic XADD(address location)
{
  int value = *location;
  *location = value + 1;
  return value
}
```

Figure A. Pseudocode for the fetch and add (XADD) atomic instruction. XADD atomically adds 1 to a value and returns the previous value. This function would be implemented as a single instruction on modern processor architectures.

```
atomic CAS(address location, value
    cmpVal, value newVal)
{
  if(*location == cmpVal) {
    *location = newVal;
    return true;
  } else return false;
}
```

Figure B. Pseudocode for the compare and swap (CAS) atomic instruction. CAS compares the value at location with cmpVal and, if they are the same, updates the value pointed to by location to newVal. This function would be implemented as a single instruction on modern processor architectures.

Figures A and B show pseudocode for two atomic instructions, fetch and add (XADD) and compare and swap (CAS), that we use for our wait-free irradiance cache. Maurice Herlihy provided a hierarchy of such primitives' effectiveness.[3] The most effective are those that can be used to implement any wait-free data structure, which Herlihy described as CAS (or load-link store-conditional instruction pairs, which are an alternative to the CAS in some architectures).

References

1. T.E. Anderson, E.D. Lazowska, and H.M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Computers*, vol. 38, no. 12, 1989, pp. 1631–1644.
2. M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann, 2008.
3. M. Herlihy, "Wait-Free Synchronization," *ACM Trans. Programming Languages and Systems*, vol. 13, no. 1, 1991, pp. 124–149.

function that generally characterizes the indirect diffuse component over space, unaffected by the high-frequency changes common with the specular component.[2] They proposed the IC to allow sparse evaluation of indirect diffuse irradiance, storing the sparsely calculated values in the IC and reusing them to extrapolate or interpolate values at nearby locations. By caching indirect diffuse irradiance samples in the framework of a distributed ray-tracing algorithm,[2] the IC allows irradiance values to be interpolated for regions in a given sample's neighborhood. This reduces rendering

Irradiance Caching

Physically based computation of the radiance reflected at a point p along a direction Θ is dictated by the rendering equation:[1]

$$L_r(p \to \Theta) = \int_{\Omega} f_r(p, \Theta \leftrightarrow \Psi) L_i(p \leftarrow \Psi) \cos(\vec{N}_p, \Psi) d_{\Omega_\Psi},$$

where $f_r(p, \Theta \leftrightarrow \Psi)$ is the bidirectional reflectance distribution function at p for directions Θ and Ψ, $L_i(p \leftarrow \Psi)$ is the incident radiance at p along Ψ, $\vec{N}_p$ is the surface normal at p, and Ω, the integration domain, is the hemisphere centered at p and oriented around $\vec{N}_p$.

Ray tracing approximates $L_r(p \to \Theta)$ by shooting rays along a number of directions Ψ, thus sampling $L_i(p \leftarrow \Psi)$ for these directions. We can usually compute certain light transport phenomena, such as specular scattering and direct illumination, with a limited number of rays. Other phenomena, because they lack directionality, require *hemisphere sampling*—that is, stochastically selecting and shooting many rays across Ω. This stochastic integration method, called Monte Carlo integration, usually accounts for much of the computation.

One such phenomenon is *indirect diffuse reflection*, the diffusely reflected radiance at p along a given direction resulting from the indirect irradiance $E(p)$. $E(p)$ is the indirect incident radiant flux per unit area at p. Accurately computing $E(p)$ requires densely sampling Ω. In a ray-tracing context, this requires shooting hundreds or thousands of rays distributed across the hemisphere while, simultaneously, carefully avoiding directions corresponding to light sources so that we include only indirect lighting—that is, light that's been reflected by at least one object.

Indirect diffuse reflections are crucial to convey a perception of realism (see Figure C), but sampling the hemisphere at all shading points results in very long rendering times, deemed unacceptable even for most offline renderings. Gregory Ward and his colleagues realized in 1988 that indirect diffuse reflection is generally a continuous smooth function over space, not affected by the high-frequency changes common with specular reflections.[2] They proposed accelerating the computation of indirect diffuse reflections by densely sampling the hemisphere at only a sparse set of shading points and interpolating the remaining ones. They store the sparsely calculated, indirect irradiance values in the irradiance cache (IC) data structure and later reuse them to extrapolate or interpolate irradiance values at nearby locations.

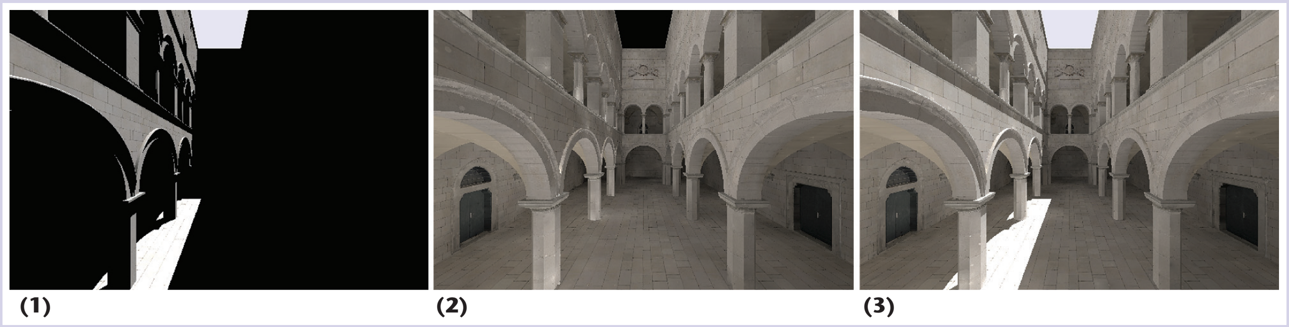


Figure C. The contribution of indirect lighting: (1) direct only, (2) indirect only, and (3) full. Without the computation of the indirect lighting, the rendered images are largely inaccurate.

time by exploiting spatial coherence. (The "Irradiance Caching" sidebar describes the motivation and mechanism behind this data structure.)

To accelerate range searches for locating valid samples in the IC, the algorithm builds an octree incrementally every time a new sample is added. Writing to the cache requires both storing the new indirect diffuse irradiance value and updating the octree. In parallel systems, each rendering process, or thread, might evaluate new indirect diffuse irradiance values and add them to the IC. To increase efficiency, all processes must share the IC, thus avoiding replicated work in which one process evaluates an irradiance value that other processes have already calculated. Ideally, the IC becomes a shared data structure, requiring some sharing mechanism to ensure that all processes can access the available data, that the data isn't corrupted, and that overheads don't compromise efficiency.

In distributed-memory systems, such as workstation clusters, each node has its own address space, resulting in multiple copies of the shared data structure that are regularly synchronized. The standard radiance distribution[6] supports a parallel renderer over a distributed system using Sun's Network File System for concurrent IC access. This approach has led to contention and can result in poor performance when using inefficient file lock managers. Roland Koholka and his colleagues broadcast IC values among processors after every 50 samples calculated at each slave.[7] David Robertson and his colleagues presented a centralized par-

We can interpolate $E(p)$ from a set $S(p)$ of previously evaluated irradiance values $E(p_i)$ at points p_i, by using a weighted average:

$$E(p) = \frac{\sum_{i \in S(p)} w_i(p) E(p_i)}{\sum_{i \in S(p)} w_i(p)},$$

where the weights

$$w_i(p) = \left(\frac{\|p - p_i\|}{R_i} + \sqrt{1 - \vec{N}_p \cdot \vec{N}_{p_i}} \right)^{-1}$$

depend on the distance between p and p_i, on the harmonic mean distance (R_i) to objects visible from p_i, and on the relative orientation of the normals at p and p_i. We can determine $S(p)$ by requiring $w_i(p)$ to be larger than the reciprocal of the maximum acceptable error, a, which is a user-supplied parameter: $S(p) = \{i : w_i(p) > 1/a\}$.

When the renderer requires indirect irradiance at any point p, it first determines $S(p)$ by querying the IC. If $S(p)$ is empty, $E(p)$ is evaluated by the Monte Carlo integration; otherwise, it's interpolated from the $E(p_i)$ belonging to $S(p)$. Querying the IC to determine $S(p)$ amounts to locating all samples p_i stored in the cache that meet the search criterion. This range search task is computationally demanding, but we can optimize it by resorting to 3D hierarchical data structures, such as octrees or *k*-d trees, and spatially ordering the IC.

By exploiting spatial coherence, the IC offers an order-of-magnitude improvement in rendering time over Monte Carlo integration. We can further improve performance when rendering animations of static scenes because the indirect diffuse irradiance remains constant, which allows reuse of the IC samples across frames.

Researchers have recently extended the IC as a stand-alone algorithm in many guises—for example, as an acceleration data structure for rendering glossy surfaces by storing radiance,[3] as a participating-media phenomenon,[4] for translucency,[5] or in conjunction with photon mapping.[6] Other extensions have exploited coherence in the temporal domain.[7–9] PDI/DreamWorks has also used similar methods to accelerate rendering.[10]

References

1. J.T. Kajiya "The Rendering Equation," *Proc. Siggraph*, ACM Press, 1986, pp. 143–150.
2. G. Ward, "A Ray Tracing Solution for Diffuse Interreflection," *Proc. Siggraph*, ACM Press, 1988, pp. 85–92.
3. J. Krivanek et al., "Radiance Caching for Efficient Global Illumination Computation," *IEEE Trans. Visualization and Computer Graphics*, vol. 11, no. 5, 2005, pp. 550–561.
4. D. Jarosz and J. Zwicker, "Radiance Caching for Participating Media," *ACM Trans. Computer Graphics*, vol. 27, no. 1, 2008, article 56.
5. S.-L. Keng, W.-Y. Lee, and J.-H. Chuang, "An Efficient Caching-Based Rendering of Translucent Materials," *The Visual Computer*, vol. 23, no. 1, 2006, pp. 59–69.
6. H.W. Jensen, *Realistic Image Synthesis Using Photon Mapping*, A K Peters, 2001.
7. M. Smyk et al., "Temporally Coherent Irradiance Caching for High Quality Animation Rendering," *Computer Graphics Forum*, vol. 24, no. 3, 2005, pp. 401–412.
8. P. Gautron, K. Bouatouch, and S. Pattanaik, "Temporal Radiance Caching," *IEEE Trans. Visualization and Computer Graphics*, vol. 13, no. 5, 2007, pp. 891–901.
9. K. Debattista et al., "Instant Caching for Interactive Global Illumination," *Computer Graphics Forum*, vol. 28, no. 8, 2009, pp. 2216–2228.
10. E. Tabellion and A. Lamorlette, "An Approximate Global Illumination System for Computer Generated Films," *ACM Trans. Graphics*, vol. 23, no. 3, 2004, pp. 469–476.

allel radiance version that sends the calculated IC values to a master process whenever a threshold is met.[8] Each slave then collects the values deposited at the master by the other slaves. Another proposal restricts diffuse irradiance evaluations to a subset of the available processors, synchronizing the IC among these at a higher frequency than with the remaining processors.[9]

We're unaware of any publication describing a data-access-control mechanism for sharing the IC among rendering threads in a shared-memory parallel system, other than our previous wait-free algorithm. The algorithm we propose here supports extendable memory for inserting an unbounded number of IC samples and for successfully inserting all new irradiance values.

Data-Access-Control Algorithms

We now present the algorithms for the three data-access-control mechanisms we evaluated in experiments.

We begin with a traditional single-threaded IC that has no access control (see Figure 1). The `IrradianceCache` data structure represents the IC; it consists of an octree of recursive nodes. The individual node is called `ICNode`. Each `ICNode` contains pointers to another eight nodes and an `ICList` storing the list of IC samples. Figure 2 shows the `ICNode` for the wait-free method. For the other methods, the `ICList` is just a single dynamic array.

The Lock-Based Irradiance Cache

The lock-based access-control algorithm (LCK)

```
IrradianceCache IC;

ComputeIndirectDiffuse() {
  //get irradiance from IC if there are valid records
  inIC = IC.getIrradiance ();
  if (!inIC) { // no valid records found
    // compute it by sampling the hemisphere
    ICsample = ComputeIrradianceRT ();
    // insert new IC sample into the octree
    IC.insert (ICsample);
  }
}

IrradianceCache::getIrradiance(Irr) {
  Irr = {0,0,0};
  <Traverse the octree>
  <verify validity of sample>
  <extrapolate irradiance; add to Irr>
  if (found) return true;
  else return false;
}

IrradianceCache::insert (ICsample) {
  // recursively traverse the octree
  // starting at root
  IC.root.insert (ICsample);
}

ICNode::insert (ICSample) {
  if (correct insertion node) {
      IClist.Add (ICsample);
  } else {
    // go deeper in the octree
    xyz = EvaluateOctant();
    if (children[xyz] == NULL)
      children[xyz] = new ICNode ();
    children[xyz].insert (ICsample);
  }
}

ICList::Add (ICsample) {
  // insert new record in head of list
  IClist.records[head++] = ICsample;
}
```

Figure 1. A traditional sequential irradiance cache (IC). This approach has no access control. It consists of an octree of recursive notes.

locks the IC whenever a read or write is made to it (see Figure 3, lines 4–6 and 12–14). However, the code responsible for hemisphere sampling, `ComputeIrradianceRT()`, isn't a critical region, so for this computation the method allows concurrent irradiance evaluation. The LCK's major disadvantage is that it serializes all accesses—both reads and writes—to the shared IC. As the number of threads increases, contention will increase, preventing performance from scaling with the degree of parallelism.

The Local-Write Irradiance Cache

An alternative approach is to have a global IC readable by all threads and an additional local IC per thread (see Figure 4). Each thread writes only to its local IC but reads from both. At certain predefined execution points, such as the end of a frame, the local ICs sequentially merge into the global IC. This synchronization uses a frame end as a barrier, effectively constituting a blocking approach to synchronization.

This approach's major drawback is that it disallows any sharing in a single frame, thus resulting in work replication. The LW algorithm has a much higher IC sample count than the other two approaches we evaluated because each thread must locally evaluate all irradiance values required by its assigned image tiles. Additionally, memory consumption is dictated by the number of threads used and the octree's complexity.

The Wait-Free Irradiance Cache

The wait-free (WF) algorithm doesn't rely on any critical sections to both read and write to the shared IC. The algorithm changes three methods from the traditional IC.

The first method is the `ICList::Add function` (see Figure 5). The insert onto the node itself takes the form of an insertion onto an array or unbounded queue. For completeness, we demonstrate how the method works for an unbounded queue. Insertion onto a fixed-size array is just a specialized case of this algorithm. The fixed-size array version is the same as the enqueue function of the Herlihy-Wing concurrent queue.[10]

The structure of `ICList` is an unbounded queue

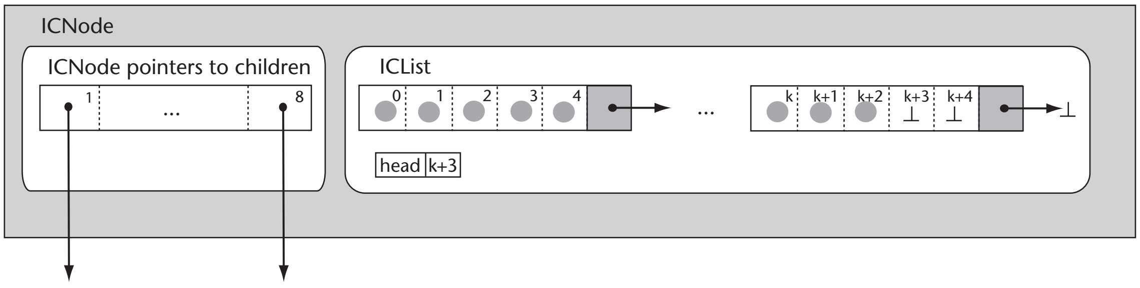


Figure 2. An `ICNode` structure for wait-free access control. For the other methods, `ICList` is just a single dynamic array that's extended whenever required.

```
ComputeIndirectDiffuse()
{
  //get irradiance from IC if there are valid records
  IC.lock();
  inIC = IC.getIrradiance (Irr);
  IC.unlock();

  if (!inIC) { // no valid records found
    // compute it by sampling the hemisphere
    ICsample = ComputeIrradianceRT ();
    // insert new IC sample into the octree
    IC.lock();
    IC.insert (ICsample);
    IC.unlock();
  }
}
```

Figure 3. A lock-based IC (LCK). This method uses busy-waiting for reading and extending the IC.

```
IrradianceCache IClocal[number threads], ICglobal;

ComputeIndirectDiffuse()
{
  //get irradiance from IC if there are valid records
  inIC = ICglobal.getIrradiance (Irr);

  if (!inIC)
    inIC = IClocal[current thread].getIrradiance ();

  if (!inIC) { // no valid records found
    // compute it by sampling the hemisphere
    ICsample = ComputeIrradianceRT ();
    // insert new sample into the local cache
    IClocal[current thread].insert (ICsample);
  }
}
```

Figure 4. A local-write IC (LW). This method maintains a separate IC per thread and synchronizes them after a single frame is rendered.

of arrays, used to maintain coherence and ensure that queue extensions aren't frequent (see Figure 2). In Figure 5, the array type is qNode, which contains an array of qNodeSize elements and a pointer to another qNode. Initially, the queue of arrays contains only one qNode; when the insertion requires a new qNode, the executing thread creates it and attaches it to the previous one. An array is always initialized with a list of NULL pointers (or some other symbol that the computation doesn't use) to denote that none of the threads has yet added an ICsample. When adding samples to an IC node, the algorithm uses the atomic fetch-and-add operator (XADD; see Figure 5, line 3). This returns a unique index into the list of records, ensuring that samples are never overwritten. Simultaneously, the thread increments the index to the next free position.

When the structure must be extended (see Figure 5, line 13), the algorithm creates a new qNode and uses a compare-and-swap instruction (CAS) to insert it onto the previous qNode (see Figure 5, line 19). If another thread hasn't yet extended the queue—that is, if the pointer is still NULL, the CAS completes successfully and the executing thread inserts the associated ICsample onto the structure (see Figure 5, line 26). However, if another thread extended the queue, the CAS will fail and this thread will discard the created qNode (see Figure 5, line 20). The thread will then insert the associated sample onto the qNode that some other thread created (otherwise, the CAS would have succeeded).

Figure 6 demonstrates three concurrent threads—*R*, *G*, and *B*—executing this method for a qNodeSize of five. This will help illustrate how ICList:Add() functions. In Figure 6a, the ICList is completely empty. In Figure 6b, *R* has just incremented the head but hasn't yet inserted the sample. *R* inserts the sample in Figure 6c. In Figure 6d, both *B* and *G* have just incremented the head but not inserted the samples. In Figure 6e, *B* hasn't inserted its sample and *R* has inserted another sample, but *G* has yet to insert the sample and is still on the same line of code as in Figure 6d.

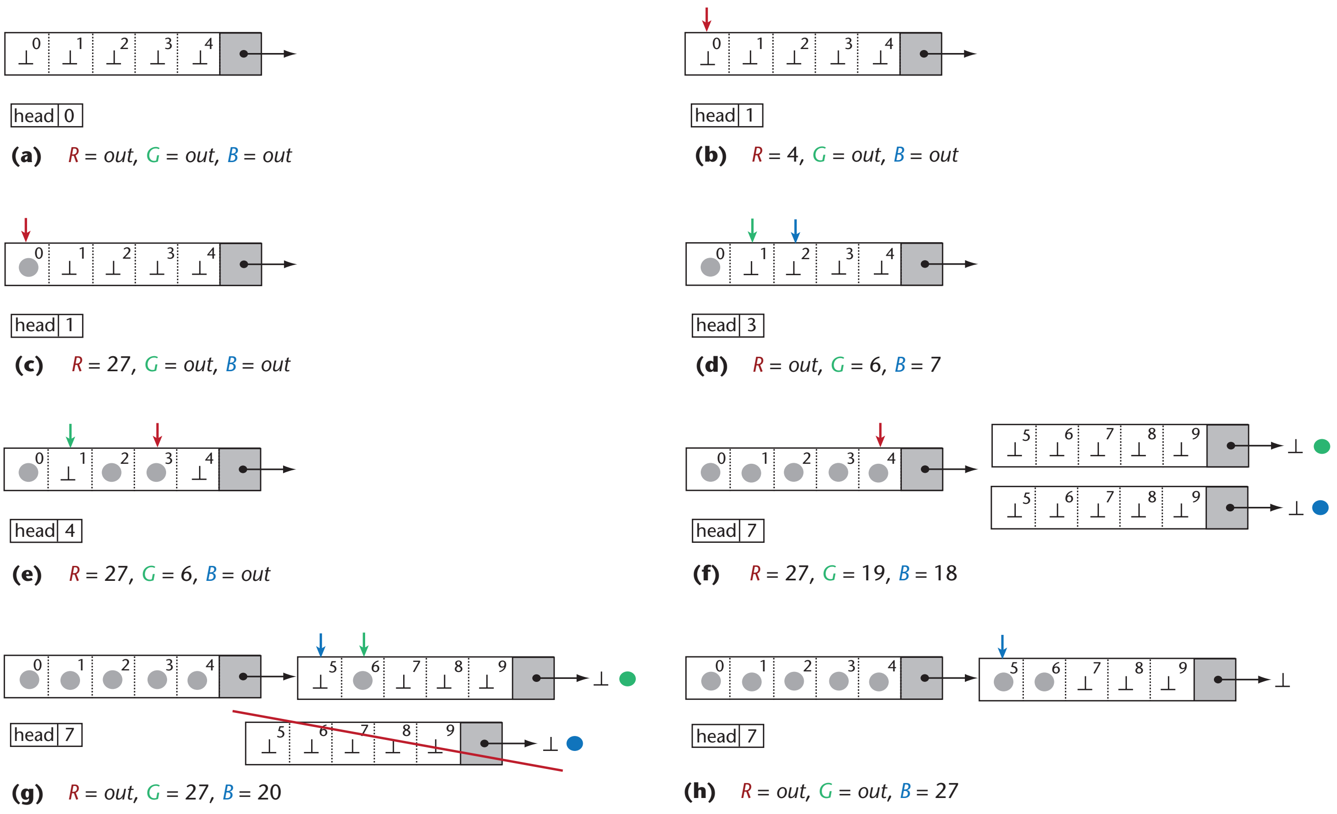
Figure 6f demonstrates the scenario of the list needing to be extended and the possible conflicts that might occur. *R* has just filled in the first qNode, and *G* and *B* are about to insert another two samples. They have, in fact, already incremented head. Because both *G* and *B* have checked that the last qNode is full and has no successor (see Figure 5, line 16), both created a new qNode. However, the CAS at line 19 in Figure 5 means that only one will succeed in attaching it to the previous qNode. In this case, at Figure 6g, we can see that *G* has succeeded and *B* is deleting the qNode it created. *G* has inserted the sample onto the new qNode. At Figure 6h, *B* inserts the sample onto the qNode that *G* had created.

The second method that's changed from the traditional IC is the insert onto the octree structure (see Figure 7). When adding a new child node to the octree, the executing thread builds the new node using a temporary pointer. Once built, the node is attached to the octree using the CAS operator (see Figure 7, line 10). The reasoning underlying this method is similar to that for ICList::Add. Either this thread creates the subtree or some other thread does; computation proceeds notwithstanding.

The final modified method is Irradiance-Cache::getIrradiance() (see Figure 8). The modifications simply reflect the structure

```
 1 ICList::Add (ICsample) {
 2   // get index of new sample in node list
 3   int index = XADD (&head);
 4   int iteration = index / qNodeSize;
 5   int pos = index % qNodeSize;
 6   qNode * tail;
 7   int count = 0;
 8
 9   // identify node — can be optimized with a local tail
10   for (tail = qHead; tail ->next != NULL && count <iteration;
11     tail = tail ->next, count++);
12
13   if (iteration > count){
14     for (int n = 0; n < iteration — count; n++) {
15       // this is where we add the new Array
16       if (tail ->next == NULL) {
17         // all entries are initialized as NULL
18         qNode * newN = new qNode;
19         if(!CAS(&tail ->next, NULL, newN))
20           delete newN;
21       }
22       tail = tail ->next;
23   }
24   // if this thread did not update
25   // some other thread must have updated
26   tail ->records[pos] = ICsample;
27   return index;
28 }
```

Figure 5. The wait-free IC `Add` method. This wait-free method ensures that IC samples can be added concurrently by different threads without blocking or busy-waiting.



head 0
(a) R = out, G = out, B = out
head 1
(b) R = 4, G = out, B = out
head 1
(c) R = 27, G = out, B = out
head 3
(d) R = out, G = 6, B = 7
head 4
(e) R = 27, G = 6, B = out
head 7
(f) R = 27, G = 19, B = 18
head 7
(g) R = out, G = 27, B = 20
head 7
(h) R = out, G = out, B = 27

Figure 6. How three threads (*R*, *G*, and *B*) would concurrently add samples to a node using our novel wait-free method. Numbers refer to the thread's location in Figure 5. *out* means that the thread isn't executing this function. ⊥ represents the `NULL` pointer.

```
ICNode::insert (ICSample) {
  if (correct insertion node)
    IClist.Add (ICsample);
  else { // go deeper in the octree

    xyz = EvaluateOctant();
    if (children[xyz]==NULL) {
      temp = new ICNode();
      // Update new branch into the octree
      if (!CAS (children[xyz], NULL, temp))
        free temp;
    }
    // irelevant to whether this thread created the subtree
    // someone must have created anyway
    // recurse the insertion of ICsample onto the subtree
    children[xyz].insert (ICsample);
  }
}
```

Figure 7. The wait-free IC `insert` method. This method ensures that the octree grows dynamically without the use of busy-waiting or blocking.

```
Irradiance Cache::getIrradiance(Irr) {
  Irr = {0,0,0};
  <Traverse the octree>
    for (qNode * tNode = qHead; tNode != NULL; tNode = tNode ->next) {
      for (i = 0; i < qNodeSize; i++)
        if (tNode ->records[i] != NULL) {
           <verify validity of sample>
           <extrapoloate irradiance; add to Irr>
        }
  }
  if (found) return true;
  else return false;
}
```

Figure 8. The wait-free IC `getIrradiance` method. The method takes into account the dynamic nature of the `ICList` construction in the wait-free IC.

changes to ICList. The method uses the fact of qNode elements being initialized to NULL. This function queries all elements that aren't NULL and uses them to calculate the irradiance, if the valid neighborhood criterion is satisfied.

The wait-free approach ensures that all threads can access the single shared IC concurrently. Our experimental results show increased execution time both when interpolating and creating IC samples, without suffering the LW approach's larger memory requirements.

Experimental Results

We obtained all the results presented here on two shared-memory systems.

One system was a dual-quad-core machine based on the Intel Xeon E5520 (Nehalem architecture), running at 2.26 GHz with 12 Gbytes of RAM. These processors include the Intel QuickPath Interconnect (which replaces the legacy front-side bus). They also support hyperthreading, enabling two threads per core and thus reporting a total of 16 logical processors to the operating system. Hyperthreading replicates certain processor resources but not the main execution units. Intel claims up to 30 percent speed improvement, compared to otherwise identical, nonhyperthreaded processors.[11]

The second system was a quad hexacore machine based on the Intel Xeon E7450 (Dunnington architecture), running at 2.40 GHz with 64 Gbytes of RAM. With 24 physical cores, this system let us evaluate the scalability of our wait-free access-control mechanism.

Both systems ran CentOS 5.2 with the Intel Compiler Suite Professional v. 11.0.

For all experiments, we used our own interactive ray tracer, which doesn't employ packetization or explicit SIMD (single instruction, multiple data) operations. The only exception is the ray-bounding volume-intersection test used to traverse the acceleration data structure, which is a bounding-volume-hierarchy implementation based on Ingo Wald and his colleagues' research.[12]

Figure 9 shows the five scenes we used. We

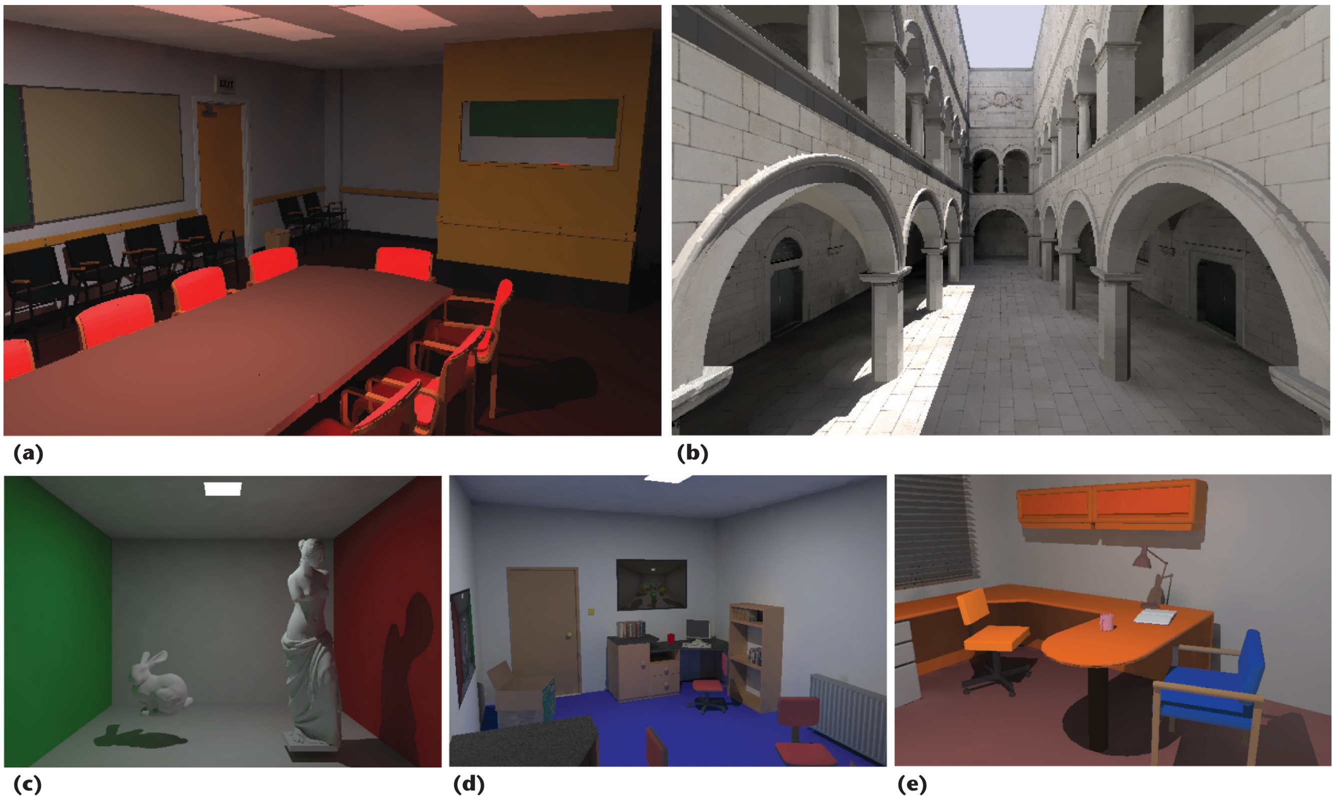


Figure 9. The five scenes used in the experiments: (a) conference room (190k polygons), (b) Sponza (66k polygons), (c) Cornell (48k polygons), (d) desk (12k polygons), and (e) office (20k polygons). The scene provide a range of geometric complexity, physical dimensions, and lighting conditions, all rendered at 640 × 480 resolution.

picked them to provide a range of geometric complexity, physical dimensions, and lighting conditions. We rendered all scenes at 640 × 480 resolution. We use the following labels for the methods: traditional sequential (TRA), lock based (LCK), local write (LW), and wait free (WF). Tables detailing the results for still images and animation are at http://doi.ieeecomputersociety.org/10.1109/MCG.2010.80.

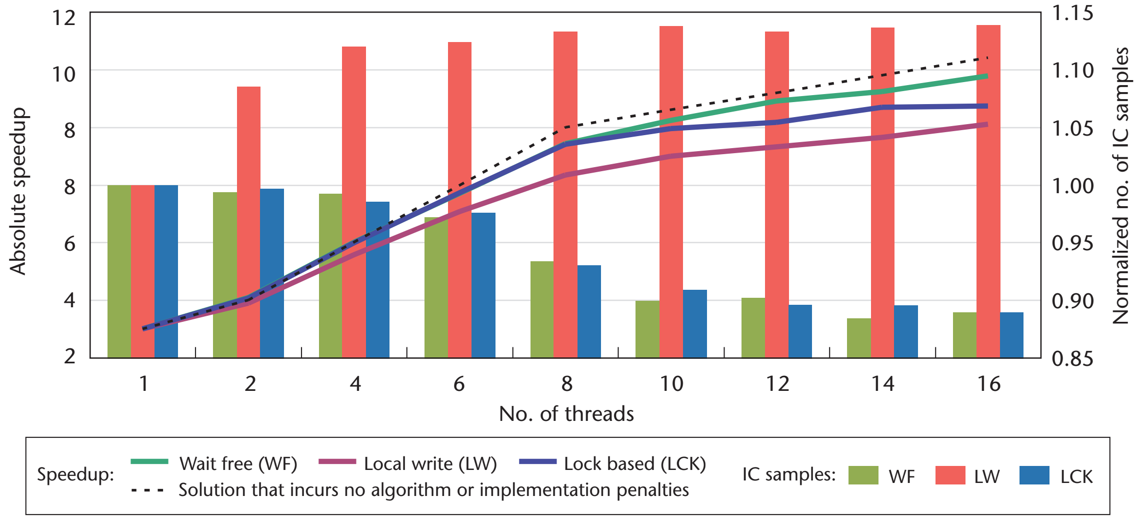
Still Images

We varied the number of threads, and thus the number of used cores, up to 16 for the Nehalem architecture and 24 for the Dunnington architecture. We obtained the results for a single thread using TRA, with no data-access-control, and computed the speedup for the different techniques with respect to the sequential timings. Each image was calculated with an empty IC to show a worst-case scenario with maximal irradiance calculations.

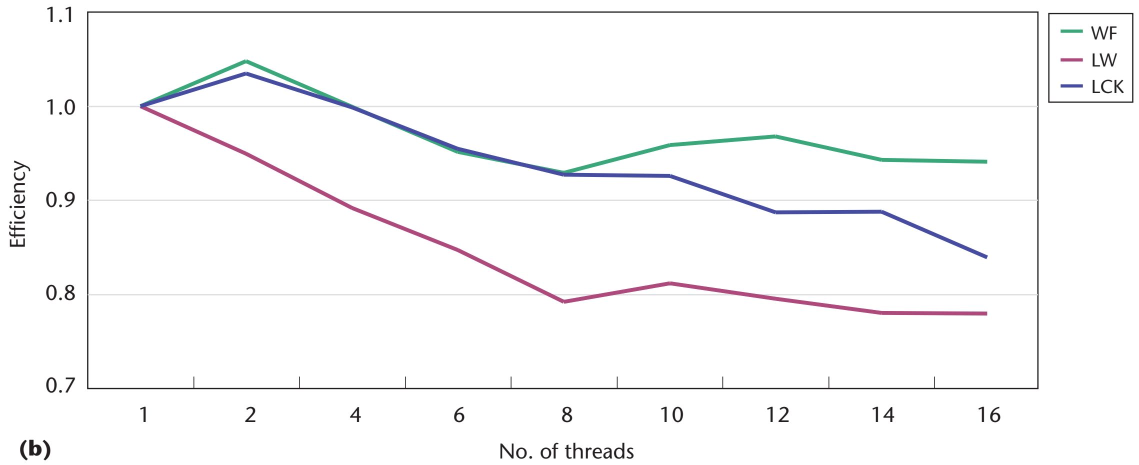
Figures 10 and 11 present the speedup, normalized number of evaluated IC samples, and efficiency for both architectures. Each metric is the average over the five scenes. We normalized the metrics with respect to the results obtained for the same scene with one single thread and the traditional approach to the IC (no data-access-control). Absolute values particular to each scene are therefore irrelevant, as long as the behavior was similar across the different scenes for each access-control mechanism. We measured worst-case standard deviations of 14.2 and 4.2 percent for, respectively, absolute speedup and the normalized number of generated IC samples. The low worst-case values indicate that we can use these averages as reliable statistics to analyze our results.

For absolute speedup, Figures 10a and 11a include a dashed line depicting the linear speedup possible if the parallel solution incurred no algorithmic or implementation penalties. For the Nehalem architecture with more than eight threads, linear speedup increased only 30 percent with each additional logical core, corresponding to Intel's claim about hyperthreading's maximum speed improvement.[11]

For all experiments with the Nehalem system and up to 14 threads in the Dunnington case, LW performed and scaled worse than the two other algorithms. This is because only one frame was rendered and the local caches merged only at the end of the frame, so no sharing actually occurred. Each thread had to evaluate all irradiance samples that projected into its assigned tiles of the image plane, leading to work replication. This is evident by the number of evaluated irradiance samples (see Figures 10a and 11a), which increased dramatically with the level of concurrency.



(a)



(b)

Figure 10. Performance for still images on the Nehalem architecture: (a) speedup and the normalized number of IC samples and (b) efficiency. (All values are averaged over the five scenes used in the experiments.) The wait-free IC presents an efficiency above 94 percent up to 16 logical (hyperthreaded) cores.

The performance difference between LCK and WF became evident as the number of threads increased: time waiting for locks grew, causing a major performance loss. The wait-free algorithm scaled much better. With fewer threads, LCK performed similarly to WF because the computation spent most of the time evaluating new irradiance samples, which isn't a critical region of the code. As the number of threads increased, the rendering performed more range searches. Because LCK serialized these searches, it incurred a performance penalty. Figures 10a and 11a clearly show that LCK's performance loss isn't due to work replication. In fact, the total number of IC samples evaluated by LCK and WF decreased above a certain number of threads. Success in finding valid samples to interpolate from depended on the order in which the threads requested and evaluated samples. Concurrent rendering of multiple image plane tiles quickly filled the IC with samples that were better distributed over object space, resulting in more successful range searches than with the sequential approach. Above a significant number of threads, LCK's serialization penalty became larger than the overhead associated with work replication, and it performed even worse than LW (see Figure 11).

Parallel algorithms seldom exhibit linear speedup because of overheads, such as load imbalance, work replication, and communication and synchronization costs. Wait-free access control can minimize the last two overheads; it exhibited almost linear speedup and, consequently, a nearly constant efficiency of 0.9 on systems up to 24 cores (see Figures 10b and 11b). With around 14 cores on the Dunnington system, LCK speedup reached an inflection point and started decreasing, showing that lock-based approaches don't scale with increasing levels of concurrency. On the other hand, WF speedup grew linearly up to 24 cores, although at

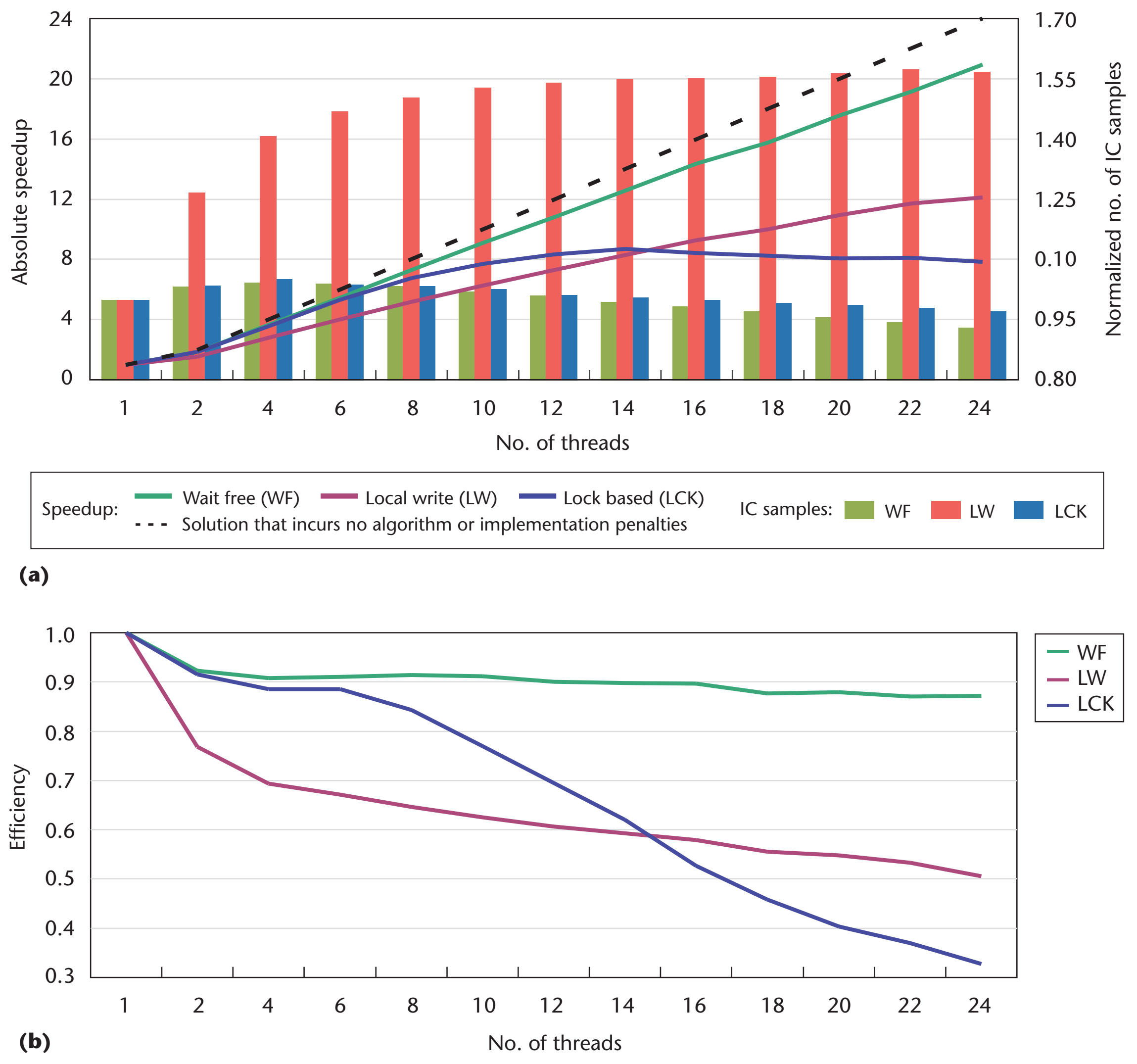


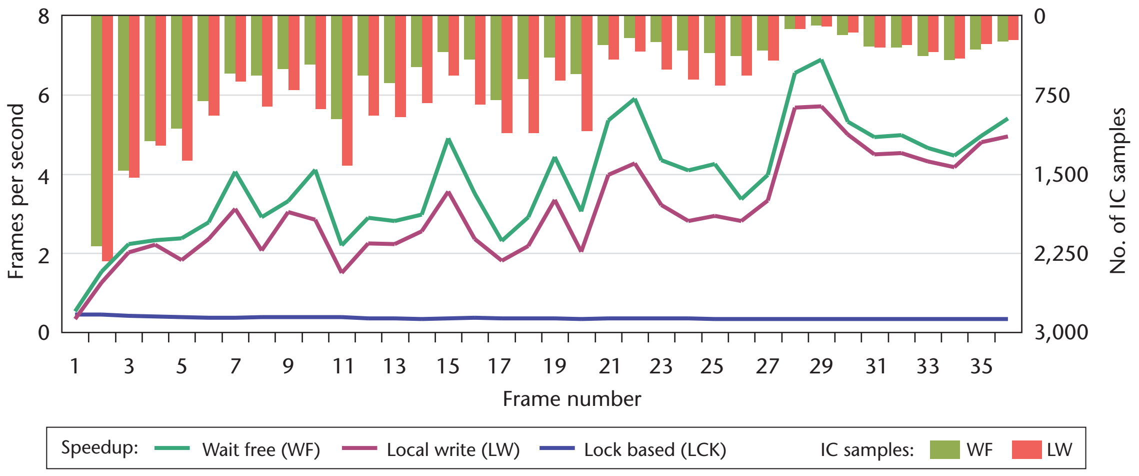
Figure 11. Performance for still images on the Dunnington architecture: (a) speedup and the normalized number of IC samples and (b) efficiency. (All values are averaged over the five scenes used in the experiments.) The wait-free IC achieves close to ideal speedup and, at 24 cores, is twice as fast as its closest rival.

a rate slightly lower than the increase in the number of cores. The WF curve's shape suggests that any eventual inflection point is still far from being reached, which demonstrates its superior scalability potential. The shared-memory parallel ray tracer also incurs overheads such as workload distribution and results gathering, which were partially responsible for the small deviation from linear speedup with WF.

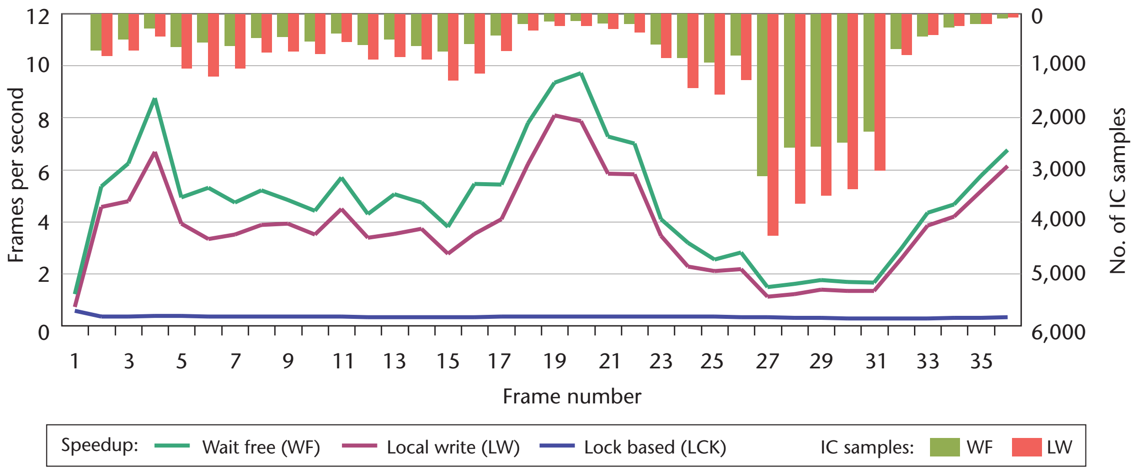
Animations

Figure 12 shows, for the Sponza and conference room scenes running on 24 cores on the Dunnington system, the frame rate and the number of IC samples evaluated per frame when running an animation of 36 frames while the camera performed a 360-degree rotation around the scene (10 degrees from frame to frame). Each frame reused previously created cache samples while simultaneously calculating new ones. This provides a performance overview of mixed evaluation and interpolation, unlike the case of the still images. For each scene, the first frame is the equivalent of one of the still images, where the cache is totally empty and all the samples must be generated.

Clearly, LCK performs worse than LW and WF. Because the IC won't be empty except for the first frame, the computation can reuse many irradiance samples. However, LCK serializes all range searches to locate these samples, thus severely affecting performance. LCK achieves the best rendering time for the first frame, suggesting that temporal reuse of previously calculated irradiance samples is worse than recalculating these values, which completely contradicts the rationale behind the IC.[6] We can thus conclude that synchronization overheads make such lock-based access mechanisms prohibitive when rendering animations of



(a)



(b)

Figure 12. Performance for animation on the Dunnington system with 24 cores, for the (a) Sponza and (b) conference room scenes. The wait-free IC outperforms the other methods for both scenes.

static scenes on highly concurrent shared-memory systems.

WF outperforms LW because the former shares irradiance samples immediately without any extra synchronization overhead associated with reading, whereas the latter doesn't share samples in a frame and so incurs extensive, costly evaluations of more indirect diffuse irradiance values. The bars in Figure 12 clearly show that, for WF and LW, performance variations from frame to frame correlate highly with the number of IC samples evaluated per frame. More important, these graphs show that WF's better results come from evaluating fewer irradiance samples, which is a consequence of efficient data sharing among threads.

In summary, LCK is penalized mostly by reading serialization, and LW is penalized by work replication. WF efficiently shares IC values while minimizing writing overheads and eliminating synchronization overheads associated with concurrent reads.

Multicore and multiprocessor systems now represent the standard form of desktop computing. Soon, such systems will likely have a degree of parallelism larger than what's available on current machines. So, the relevance of efficient, scalable, and reliable shared data structures for maximizing performance is continually increasing. Wait-free data structures offer an alternative to traditional locking and blocking approaches and enable traditional graphics algorithms to exploit modern hardware. The shared-memory IC we've demonstrated shows such techniques' potential. Our algorithm has let us achieve close to interactive rates for ray tracing with global illumination. We hope our solution will motivate similar parallel methods in other computer graphics areas.

Acknowledgments

Partial funding for this work came from the Portuguese Foundation for Science and Technology's Project IGIDE

(Interactive Global Illumination within Dynamic Environments), grant PT-DC/EIA/65965/2006, and from the United Kingdom's Engineering and Physical Sciences Research Council, grant EP/D069874/2. Greg Ward provided the office and conference scenes from the Radiance package, and Stanford's Graphics Group provided the bunny model from the Stanford 3D Repository. Alberto Proença granted us access to the two multicore systems.

References

1. M. Herlihy, "Technical Perspective: Highly Concurrent Data Structures," *Comm. ACM*, vol. 52, no. 5, 2009, p. 99.
2. G. Ward, "A Ray Tracing Solution for Diffuse Interreflection," *Proc. Siggraph*, ACM Press, 1988, pp. 85–92.
3. M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," *Proc. 23rd Int'l Conf. Distributed Computing Systems* (ICDCS 03), IEEE CS Press, 2003, pp. 522–529.
4. P. Dubla et al., "Wait-Free Shared-Memory Irradiance Cache," *Proc. Eurographics Symp. Parallel Graphics and Visualization* (ESPGV 09), Eurographics Assoc., 2009, pp. 57–64.
5. I. Wald et al., "State of the Art in Ray Tracing Animated Scenes," *Eurographics 2007 State of the Art Reports*, Eurographics Assoc., 2007, pp. 89–116.
6. G. Ward, "The Radiance Lighting Simulation and Rendering System," *Proc. Siggraph*, ACM Press, 1994, pp. 459–472.
7. R. Koholka, H. Mayer, and A. Goller, "MPI-Parallelized Radiance on SGI CoW and SMP," *Proc. 4th Int'l ACPC Conf.* (ParNum 99), Springer, 1999, pp. 549–558.
8. D. Robertson et al., "Parallelization of Radiance for Real Time Interactive Lighting Visualization Walkthroughs," *Proc. 1999 ACM/IEEE Conf. Supercomputing*, ACM Press, 1999, article 61.
9. K. Debattista, L.P. Santos, and A. Chalmers, "Accelerating the Irradiance Cache through Parallel Component-Based Rendering," *Proc. Eurographics Symp. Parallel Graphics and Visualization*, Eurographics Assoc., 2006, pp. 27–34.
10. M.P. Herlihy and J.M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Trans. Programming Languages and Systems*, vol. 12, no. 3, 1990, pp. 463–492.
11. D. Marr et al., "Hyper-threading Technology Architecture and Microarchitecture," *Intel Technology J.*, vol. 6, no. 1, 2002, pp. 4–15.
12. I. Wald, S. Boulos, and P. Shirley, "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies," *ACM Trans. Graphics*, vol. 26, no. 1, 2007, article 6.

Kurt Debattista *is an assistant professor at the University of Warwick's International Digital Laboratory, WMG. His research focuses on high-fidelity rendering, parallel computing, high-dynamic-range imaging, and serious games. Debattista has a PhD in computer science from the University of Bristol. Contact him at k.debattista@warwick.ac.uk.*

Piotr Dubla *is a graphics programmer at Rockstar North, a game studio, while completing his PhD at the University of Warwick's International Digital Laboratory, WMG. His research interests include interactive ray tracing, global illumination, offline physically based rendering, and realistic simulations of light transport as well as parallel processing and data structures for computer graphics and procedural content generation. Dubla has a BSc Hons (with distinction) in computer science from the University of Cape Town. Contact him at p.b.dubla@wariwck.ac.uk.*

Luís Paulo Peixoto dos Santos *is an auxiliary professor in Universidade do Minho's Department of Computer Science. His research interests include high-fidelity interactive rendering and parallel processing. Santos has a PhD in parallel processing from Universidade do Minho. Contact him at psantos@di.uminho.pt.*

Alan Chalmers *is a professor at the University of Warwick's International Digital Laboratory, WMG. His research interests include high-fidelity graphics, multisensory perception, virtual archaeology, and parallel rendering. Chalmers has a PhD in computer science from the University of Bristol. He is the honorary president of Afrigraph and a former vice president of ACM Siggraph. Contact him at a.g.chalmers@warwick.ac.uk.*