Memory part 8: Future technologies

November 14, 2007
This article was contributed by Ulrich Drepper

[*Editor's note: Here, at last, is the final segment of Ulrich Drepper's "What every programmer should know about memory." This eight-part series began back in September. The conclusion of this document looks at how future technologies may help to improve performance as the memory bottleneck continues to worsen.*

We would like to thank Ulrich one last time for giving LWN the opportunity to help shape this document and bring it to our readers. Ulrich plans to post the entire thing in PDF format sometime in the near future; we'll carry an announcement when that happens.]

8 Upcoming Technology

In the preceding sections about multi-processor handling we have seen that significant performance problems must be expected if the number of CPUs or cores is scaled up. But this scaling-up is exactly what has to be expected in the future. Processors will get more and more cores, and programs must be ever more parallel to take advantage of the increased potential of the CPU, since single-core performance will not rise as quickly as it used to.

8.1 The Problem with Atomic Operations

Synchronizing access to shared data structures is traditionally done in two ways:

- through mutual exclusion, usually by using functionality of the system runtime to achieve just that;
- by using lock-free data structures.

The problem with lock-free data structures is that the processor has to provide primitives which can perform the entire operation atomically. This support is limited. On most architectures support is limited to atomically read and write a word. There are two basic ways to implement this (see Section 6.4.2):

- using atomic compare-and-exchange (CAS) operations;
- using a load lock/store conditional (LL/SC) pair.

It can be easily seen how a CAS operation can be implemented using LL/SC instructions. This makes CAS operations the building block for most atomic operations and lock free data structures.

Some processors, notably the x86 and x86-64 architectures, provide a far more elaborate set of atomic operations. Many of them are optimizations of the CAS operation for specific purposes. For instance, atomically adding a value to a memory location can be implemented using CAS and LL/SC operations, but the native support for atomic increments on x86/x86-64 processors is faster. It is important for programmers to know about these operations, and the intrinsics which make them available when programming, but that is nothing new.

The extraordinary extension of these two architectures is that they have double-word CAS (DCAS) operations. This is significant for some applications but not all (see [dcas]). As an example of how

DCAS can be used, let us try to write a lock-free array-based stack/LIFO data structure. A first attempt using gcc's intrinsics can be seen in Figure 8.1.

```
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  n->c = top;
  top = n;
}
struct elem *pop(void) {
  struct elem *res = top;
  if (res != NULL)
    top = res->c;
  return res;
}
```

Figure 8.1: Not Thread-Safe LIFO

This code is clearly not thread-safe. Concurrent accesses in different threads will modify the global variable `top` without consideration of other thread's modifications. Elements could be lost or removed elements can magically reappear. It is possible to use mutual exclusion but here we will try to use only atomic operations.

The first attempt to fix the problem uses CAS operations when installing or removing list elements. The resulting code looks like Figure 8.2.

```
#define CAS __sync_bool_compare_and_swap
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  do
    n->c = top;
  while (!CAS(&top, n->c, n));
}
struct elem *pop(void) {
  struct elem *res;
  while ((res = top) != NULL)
    if (CAS(&top, res, res->c))
      break;
  return res;
}
```

Figure 8.2: LIFO using CAS

At first glance this looks like a working solution. `top` is never modified unless it matches the element which was at the top of the LIFO when the operation started. But we have to take concurrency at all levels into account. It might be that another thread working on the data structure is scheduled at the worst possible moment. One such case here is the so-called ABA problem. Consider what happens if a second thread is scheduled right before the CAS operation in `pop` and it performs the following operation:

1. `l = pop()`
2. `push(newelem)`
3. `push(l)`

The end effect of this operation is that the former top element of the LIFO is back at the top but the second element is different. Back in the first thread, because the top element is unchanged, the CAS operation will succeed. But the value `res->c` is not the right one. It is a pointer to the second element of the original LIFO and not `newelem`. The result is that this new element is lost.

In the literature [lockfree] you find suggestions to use a feature found on some processors to work around this problem. Specifically, this is about the ability of the x86 and x86-64 processors to perform DCAS operations. This is used in the third incarnation of the code in Figure 8.3.

```
#define CAS __sync_bool_compare_and_swap
struct elem {
  data_t d;
  struct elem *c;
};
struct lifo {
  struct elem *top;
  size_t gen;
} l;
void push(struct elem *n) {
  struct lifo old, new;
  do {
    old = l;
    new.top = n->c = old.top;
    new.gen = old.gen + 1;
  } while (!CAS(&l, old, new));
}
struct elem *pop(void) {
  struct lifo old, new;
  do {
    old = l;
    if (old.top == NULL) return NULL;
    new.top = old.top->c;
    new.gen = old.gen + 1;
  } while (!CAS(&l, old, new));
  return old.top;
}
```

Figure 8.3: LIFO using double-word CAS

Unlike the other two examples, this is (currently) pseudo-code since gcc does not grok the use of structures in the CAS intrinsics. Regardless, the example should be sufficient understand the approach. A generation counter is added to the pointer to the top of the LIFO. Since it is changed on every operation, `push` or `pop`, the ABA problem described above is no longer a problem. By the time the first thread is resuming its work by actually exchanging the `top` pointer, the generation counter has been incremented three times. The CAS operation will fail and, in the next round of the loop, the correct first and second element of the LIFO are determined and the LIFO is not corrupted. Voilà.

Is this really the solution? The authors of [lockfree] certainly make it sound like it and, to their credit, it should be mentioned that it is possible to construct data structures for the LIFO which would permit using the code above. But, in general, this approach is just as doomed as the previous one. We still have concurrency problems, just now in a different place. Let us assume a thread executes `pop` and is interrupted after the test for `old.top == NULL`. Now a second thread uses `pop` and receives ownership of the previous first element of the LIFO. It can do anything with it, including changing all values or, in case of dynamically allocated elements, freeing the memory.

Now the first thread resumes. The `old` variable is still filled with the previous top of the LIFO. More specifically, the `top` member points to the element popped by the second thread. In `new.top = old.top->c` the first thread dereferences a pointer in the element. But the element this pointer references might have been freed. That part of the address space might be inaccessible and the process could crash. This cannot be allowed for a generic data type implementation. Any fix for this problem is terribly expensive: memory must never be freed, or at least it must be verified that no thread is referencing the memory anymore before it is freed. Given that lock-free data structures are supposed to be faster and more concurrent, these additional requirements completely destroy any advantage. In languages which support it, memory handling through garbage collection can solve the problem, but this comes with its price.

The situation is often worse for more complex data structures. The same paper cited above also describes a FIFO implementation (with refinements in a successor paper). But this code has all the same problems. Because CAS operations on existing hardware (x86, x86-64) are limited to modifying two words which are consecutive in memory, they are no help at all in other common situations. For instance, atomically adding or removing elements anywhere in a double-linked list is not possible. {*As a side note, the developers of the IA-64 did* not *include this feature. They allow* comparing *two words, but replacing only one.*}

The problem is that more than one memory address is generally involved, and only if none of the values of these addresses is changed concurrently can the entire operation succeed. This is a well-known concept in database handling, and this is exactly where one of the most promising proposals to solve the dilemma comes from.

8.2 Transactional Memory

In their groundbreaking 1993 paper [transactmem] Herlihy and Moss propose to implement transactions for memory operations in hardware since software alone cannot deal with the problem efficiently. Digital Equipment Corporation, at that time, was already battling with scalability problems on their high-end hardware, which featured a few dozen processors. The principle is the same as for database transactions: the result of a transaction becomes visible all at once or the transaction is aborted and all the values remain unchanged.

This is where memory comes into play and why the previous section bothered to develop algorithms which use atomic operations. Transactional memory is meant as a replacement for—and extension of—atomic operations in many situations, especially for lock-free data structures. Integrating a transaction system into the processor sounds like a terribly complicated thing to do but, in fact, most processors, to some extent, already have something similar.

The LL/SC operations implemented by some processors form a transaction. The SC instruction aborts or commits the transaction based on whether the memory location was touched or not. Transactional memory is an extension of this concept. Now, instead of a simple pair of instructions, multiple instructions take part in the transaction. To understand how this can work, it is worthwhile to first see how LL/SC instructions can be implemented. {*This does not mean it is actually implemented like this.*}

8.2.1 Load Lock/Store Conditional Implementation

If the LL instruction is issued, the value of the memory location is loaded into a register. As part of that operation, the value is loaded into L1d. The SC instruction later can only succeed if this value has not been tampered with. How can the processor detect this? Looking back at the description of the MESI protocol in Figure 3.18 should make the answer obvious. If another processor changes the value of the memory location, the copy of the value in L1d of the first processor must be revoked. When the SC instruction is executed on the first processor, it will find it has to load the value again into L1d. This is something the processor must already detect.

There are a few more details to iron out with respect to context switches (possible modification on the same processor) and accidental reloading of the cache line after a write on another processor. This is nothing that policies (cache flush on context switch) and extra flags, or separate cache lines for LL/SC instructions, cannot fix. In general, the LL/SC implementation comes almost for free with the implementation of a cache coherence protocol like MESI.

8.2.2 Transactional Memory Operations

For transactional memory to be generally useful, a transaction must not be finished with the first store instruction. Instead, an implementation should allow a certain number of load and store operations; this means we need separate commit and abort instructions. In a bit we will see that we

need one more instruction which allows checking on the current state of the transaction and whether it is already aborted or not.

There are three different memory operations to implement:

- Read memory
- Read memory which is written to later
- Write memory

When looking at the MESI protocol it should be clear how this special second type of read operation can be useful. The normal read can be satisfied by a cache line in the `E' and `S' state. The second type of read operation needs a cache line in state `E'. Exactly why the second type of memory read is necessary can be glimpsed from the following discussion, but, for a more complete description, the interested reader is referred to literature about transactional memory, starting with [transactmem].

In addition, we need transaction handling which mainly consists of the commit and abort operation we are already familiar with from database transaction handling. There is one more operation, though, which is optional in theory but required for writing robust programs using transactional memory. This instruction lets a thread test whether the transaction is still on track and can (perhaps) be committed later, or whether the transaction already failed and will in any case be aborted.

We will discuss how these operations actually interact with the CPU cache and how they match to bus operation. But before we do that we take a look at some actual code which uses transactional memory. This will hopefully make the remainder of this section easier to understand.

8.2.3 Example Code Using Transactional Memory

For the example we revisit our running example and show a LIFO implementation which uses transactional memory.

```
struct elem {
  data_t d;
  struct elem *c;
};
struct elem *top;
void push(struct elem *n) {
  while (1) {
    n->c = LTX(top);
    ST(&top, n);
    if (COMMIT())
      return;
    ... delay ...
  }
}
struct elem *pop(void) {
  while (1) {
    struct elem *res = LTX(top);
    if (VALIDATE()) {
      if (res != NULL)
        ST(&top, res->c);
      if (COMMIT())
        return res;
    }
    ... delay ...
  }
}
```

Figure 8.4: LIFO Using Transactional Memory

This code looks quite similar to the not-thread-safe code, which is an additional bonus as it makes writing code using transactional memory easier. The new parts of the code are the `LTX`, `ST`, `COMMIT`, and `VALIDATE` operations. These four operations are the way to request accesses to transactional memory. There is actually one more operation, `LT`, which is not used here. `LT` requests non-exclusive read

access, LTX requests exclusive read access, and ST is a store into transactional memory. The VALIDATE operation is the operation which checks whether the transaction is still on track to be committed. It returns true if this transaction is still OK. If the transaction is already marked as aborting, it will be actually aborted and the next transactional memory instruction will start a new transaction. For this reason, the code uses a new if block in case the transaction is still going on.

The COMMIT operation finishes the transaction; if the transaction is finished successfully the operation returns true. This means that this part of the program is done and the thread can move on. If the operation returns a false value, this usually means the whole code sequence must be repeated. This is what the outer while loop is doing here. This is not absolutely necessary, though, in some cases giving up on the work is the right thing to do.

The interesting point about the LT, LTX, and ST operations is that they can fail without signaling this failure in any direct way. The way the program can request this information is through the VALIDATE or COMMIT operation. For the load operation, this can mean that the value actually loaded into the register might be bogus; that is why it is necessary in the example above to use VALIDATE before dereferencing the pointer. In the next section, we will see why this is a wise choice for an implementation. It might be that, once transactional memory is actually widely available, the processors will implement something different. The results from [transactmem] suggest what we describe here, though.

The push function can be summarized as this: the transaction is started by reading the pointer to the head of the list. The read requests exclusive ownership since, later in the function, this variable is written to. If another thread has already started a transaction, the load will fail and mark the still-born transaction as aborted; in this case, the value actually loaded might be garbage. This value is, regardless of its status, stored in the next field of the new list member. This is fine since this member is not yet in use, and it is accessed by exactly one thread. The pointer to the head of the list is then assigned the pointer to the new element. If the transaction is still OK, this write can succeed. This is the normal case, it can only fail if a thread uses some code other than the provided push and pop functions to access this pointer. If the transaction is already aborted at the time the ST is executed, nothing at all is done. Finally, the thread tries to commit the transaction. If this succeeds the work is done; other threads can now start their transactions. If the transaction fails, it must be repeated from the beginning. Before doing that, however, it is best to insert an delay. If this is not done the thread might run in a busy loop (wasting energy, overheating the CPU).

The pop function is slightly more complex. It also starts with reading the variable containing the head of the list, requesting exclusive ownership. The code then immediately checks whether the LTX operation succeeded or not. If not, nothing else is done in this round except delaying the next round. If the top pointer was read successfully, this means its state is good; we can now dereference the pointer. Remember, this was exactly the problem with the code using atomic operations; with transactional memory this case can be handled without any problem. The following ST operation is only performed when the LIFO is not empty, just as in the original, thread-unsafe code. Finally the transaction is committed. If this succeeds the function returns the old pointer to the head; otherwise we delay and retry. The one tricky part of this code is to remember that the VALIDATE operation aborts the transaction if it has already failed. The next transactional memory operation would start a new transaction and, therefore, we must skip over the rest of the code in the function.

How the delay code works will be something to see when implementations of transactional memory are available in hardware. If this is done badly system performance might suffer significantly.

8.2.4 Bus Protocol for Transactional Memory

Now that we have seen the basic principles behind transactional memory, we can dive into the details of the implementation. Note that this is *not* based on actual hardware. It is based on the

original design of transactional memory and knowledge about the cache coherency protocol. Some details are omitted, but it still should be possible to get insight into the performance characteristics.

Transactional memory is not actually implemented as separate memory; that would not make any sense given that transactions on any location in a thread's address space are wanted. Instead, it is implemented at the first cache level. The implementation could, in theory, happen in the normal L1d but, as [transactmem] points out, this is not a good idea. We will more likely see the transaction cache implemented in parallel to L1d. All accesses will use the higher level cache in the same way they use L1d. The transaction cache is likely much smaller than L1d. If it is fully associative its size is determined by the number of operations a transaction can comprise. Implementations will likely have limits for the architecture and/or specific processor version. One could easily imagine a transaction cache with 16 elements or even less. In the above example we only needed one single memory location; algorithms with a larger transaction working sets get very complicated. It is possible that we will see processors which support more than one active transaction at any one time. The number of elements in the cache then multiplies, but it is still small enough to be fully associative.

The transaction cache and L1d are exclusive. That means a cache line is in, at most, one of the caches but never in both. Each slot in the transaction cache is in, at any one time, one of the four MESI protocol states. In addition to this, a slot has a transaction state. The states are as follows (names according to [transactmem]):

EMPTY
: the cache slot contains no data. The MESI state is always 'I'.

NORMAL
: the cache slot contains committed data. The data could as well exist in L1d. The MESI state can be 'M', 'E', and 'S'. The fact that the 'M' state is allowed means that transaction commits do *not* force the data to be written into the main memory (unless the memory region is declared as uncached or write-through). This can significantly help to increase performance.

XABORT
: the cache slot contains data which is discarded on abort. This is obviously the opposite of XCOMMIT. All the data created during a transaction is kept in the transaction cache, nothing is written to main memory before a commit. This limits the maximum transaction size but it means that, beside the transaction cache, no other memory has to be aware of the XCOMMIT/XABORT duality for a single memory location. The possible MESI states are 'M', 'E', and 'S'.

XCOMMIT
: the cache slot contains data which is discarded on commit. This is a possible optimization processors could implement. If a memory location is changed using a transaction operation, the old content cannot be just dropped: if the transaction fails the old content needs to be restored. The MESI states are the same as for XABORT. One difference with regard to XABORT is that, if the transaction cache is full, any XCOMMIT entries in the 'M' state could be written back to memory and then, for all states, discarded.

When an `LT` operation is started, the processor allocates two slots in the cache. Victims are chosen by first looking for NORMAL slots for the address of the operation, i.e., a cache hit. If such an entry is found, a second slot is located, the value copied, one entry is marked XABORT, and the other one is marked XCOMMIT.

If the address is not already cached, EMPTY cache slots are located. If none can be found, NORMAL slots are looked for. The old content must then be flushed to memory if the MESI state is 'M'. If no NORMAL slot is available either, it is possible to victimize XCOMMIT entries. This is likely going to be

an implementation detail, though. The maximum size of a transaction is determined by the size of the transaction cache, and, since the number of slots which are needed for each operation in the transaction is fixed, the number of transactions can be capped before having to evict XCOMMIT entries.

If the address is not found in the transactional cache, a T_READ request is issued on the bus. This is just like the normal READ bus request, but it indicates that this is for the transactional cache. Just like for the normal READ request, the caches in all other processors first get the chance to respond. If none does the value is read from the main memory. The MESI protocol determines whether the state of the new cache line is 'E' or 'S'. The difference between T_READ and READ comes into play when the cache line is currently in use by an active transaction on another processor or core. In this case the T_READ operation plainly fails, no data is transmitted. The transaction which generated the T_READ bus request is marked as failed and the value used in the operation (usually a simple register load) is undefined. Looking back to the example, we can see that this behavior does not cause problems if the transactional memory operations are used correctly. Before a value loaded in a transaction is used, it must be verified with `VALIDATE`. This is, in almost no cases, an extra burden. As we have seen in the attempts to create a FIFO implementation using atomic operations, the check which we added is the one missing feature which would make the lock-free code work.

The `LTX` operation is almost identical to `LT`. The one difference is that the bus operation is T_RFO instead of T_READ. T_RFO, like the normal RFO bus request, requests exclusive ownership of the cache line. The state of the resulting cache line is 'E'. Like the T_READ bus request, T_RFO can fail, in which case the used value is undefined, too. If the cache line is already in the local transaction cache with 'M' or 'E' state, nothing has to be done. If the state in the local transaction cache is 'S' the bus request has to go out to invalidate all other copies.

The `ST` operation is similar to `LTX`. The value is first made available exclusively in the local transaction cache. Then the `ST` operation makes a copy of the value into a second slot in the cache and marks the entry as XCOMMIT. Lastly, the other slot is marked as XABORT and the new value is written into it. If the transaction is already aborted, or is newly aborted because the implicit `LTX` fails, nothing is written.

Neither the `VALIDATE` nor `COMMIT` operations automatically and implicitly create bus operations. This is the huge advantage transactional memory has over atomic operations. With atomic operations, concurrency is made possible by writing changed values back into main memory. If you have read this document thus far, you should know how expensive this is. With transactional memory, no accesses to the main memory are forced. If the cache has no EMPTY slots, current content must be evicted, and for slots in the 'M' state, the content must be written to main memory. This is not different from regular caches, and the write-back can be performed without special atomicity guarantees. If the cache size is sufficient, the content can survive for a long time. If transactions are performed on the same memory location over and over again, the speed improvements can be astronomical since, in the one case, we have one or two main memory accesses in each round while, for transactional memory, all accesses hit the transactional cache, which is as fast as L1d.

All the `VALIDATE` and `COMMIT` operations do for an aborted transaction is to mark the cache slots marked XABORT as empty and mark the XCOMMIT slots as NORMAL. Similarly, when `COMMIT` successfully finishes a transaction, the XCOMMIT slots are marked empty and the XABORT slots are marked NORMAL. These are very fast operations on the transaction cache. No explicit notification to other processors which want to perform transactions happens; those processors just have to keep trying. Doing this efficiently is another matter. In the example code above we simply have `...delay...` in the appropriate place. We might see actual processor support for delaying in a useful way.

To summarize, transactional memory operations cause bus operation only when a new transaction is started and when a new cache line, which is not already in the transaction cache, is added to a still-

successful transaction. Operations in aborted transactions do not cause bus operations. There will be no cache line ping-pong due to multiple threads trying to use the same memory.

8.2.5 Other Considerations

In Section 6.4.2, we already discussed how the `lock` prefix, available on x86 and x86-64, can be used to avoid the coding of atomic operations in some situations. The proposed tricks falls short, though, when there are multiple threads in use which do not contend for the same memory. In this case, the atomic operations are used unnecessarily. With transactional memory this problem goes away. The expensive RFO bus requests are issued only if memory is used on different CPUs concurrently or in succession; this is only the case when they are needed. It is almost impossible to do any better.

The attentive reader might have wondered about delays. What is the expected worst case scenario? What if the thread with the active transaction is descheduled, or if it receives a signal and is possibly terminated, or decides to use `siglongjmp` to jump to an outer scope? The answer to this is: the transaction will be aborted. It is possible to abort a transaction whenever a thread makes a system call or receives a signal (i.e., a ring level change occurs). It might also be that aborting the transaction is part of the OS's duties when performing system calls or handling signals. We will have to wait until implementations become available to see what is actually done.

The final aspect of transactional memory which should be discussed here is something which people might want to think about even today. The transaction cache, like other caches, operates on cache lines. Since the transaction cache is an exclusive cache, using the same cache line for transactions and non-transaction operation will be a problem. It is therefore important to

- move non-transactional data off of the cache line
- have separate cache lines for data used in separate transactions

The first point is not new, the same effort will pay off for atomic operations today. The second is more problematic since today objects are hardly ever aligned to cache lines due to the associated high cost. If the data used, along with the words modified using atomic operations, is on the same cache line, one less cache line is needed. This does not apply to mutual exclusion (where the mutex object should always have its own cache line), but one can certainly make cases where atomic operations go together with other data. With transactional memory, using the cache line for two purposes will most likely be fatal. Every normal access to data {*From the cache line in question. Access to arbitrary other cache lines does not influence the transaction.*} would remove the cache line from the transactional cache, aborting the transaction in the process. Cache alignment of data objects will be in future not only a matter of performance but also of correctness.

It is possible that transactional memory implementations will use more precise accounting and will, as a result, not suffer from normal accesses to data on cache lines which are part of a transaction. This requires a lot more effort, though, since then the MESI protocol information is not sufficient anymore.

8.3 Increasing Latency

One thing about future development of memory technology is almost certain: latency will continue to creep up. We already discussed, in Section 2.2.4, that the upcoming DDR3 memory technology will have higher latency than the current DDR2 technology. FB-DRAM, if it should get deployed, also has potentially higher latency, especially when FB-DRAM modules are daisy-chained. Passing through the requests and results does not come for free.

The second source of latency is the increasing use of NUMA. AMD's Opterons are NUMA machines if they have more than one processor. There is some local memory attached to the CPU with its own memory controller but, on SMP motherboards, the rest of the memory has to be accessed through

the Hypertransport bus. Intel's CSI technology will use almost the same technology. Due to per-processor bandwidth limitations and the requirement to keep (for instance) multiple 10Gb/s Ethernet ports busy, multi-socket motherboards will not vanish, even if the number of cores per socket increases.

A third source of latency are co-processors. We thought that we got rid of them after math co-processors for commodity processors were no longer necessary at the beginning of the 1990's, but they are making a comeback. Intel's Geneseo and AMD's Torrenza are extensions of the platform to allow third-party hardware developers to integrate their products into the motherboards. I.e., the co-processors will not have to sit on a PCIe card but, instead, are positioned much closer to the CPU. This gives them more bandwidth.

IBM went a different route (although extensions like Intel's and AMD's are still possible) with the Cell CPU. The Cell CPU consists, beside the PowerPC core, of 8 Synergistic Processing Units (SPUs) which are specialized processors mainly for floating-point computation.

What co-processors and SPUs have in common is that they, most likely, have even slower memory logic than the real processors. This is, in part, caused by the necessary simplification: all the cache handling, prefetching etc is complicated, especially when cache coherency is needed, too. High-performance programs will increasingly rely on co-processors since the performance differences can be dramatic. Theoretical peak performance for a Cell CPU is 210 GFLOPS, compared to 50-60 GFLOPS for a high-end CPU. The Graphics Processing Units (GPUs, processors on graphics cards) in use today achieve even higher numbers (north of 500 GFLOPS) and those could probably, with not too much effort, be integrated into the Geneseo/Torrenza systems.

As a result of all these developments, a programmer must conclude that prefetching will become ever more important. For co-processors it will be absolutely critical. For CPUs, especially with more and more cores, it is necessary to keep the FSB busy all the time instead of piling on the requests in batches. This requires giving the CPU as much insight into future traffic as possible through the efficient use of prefetching instructions.

8.4 Vector Operations

The multi-media extensions in today's mainstream processors implement vector operations only in a limited fashion. Vector instructions are characterized by large numbers of operations which are performed together. Compared with scalar operations, this can be said about the multi-media instructions, but it is a far cry from what vector computers like the Cray-1 or vector units for machines like the IBM 3090 did.

To compensate for the limited number of operations performed for one instruction (four `float` or two `double` operations on most machines) the surrounding loops have to be executed more often. The example in Section 9.1 shows this clearly, each cache line requires `SM` iterations.

With wider vector registers and operations, the number of loop iterations can be reduced. This results in more than just improvements in the instruction decoding etc.; here we are more interested in the memory effects. With a single instruction loading or storing more data, the processor has a better picture about the memory use of the application and does not have to try to piece together the information from the behavior of individual instructions. Furthermore, it becomes more useful to provide load or store instructions which do not affect the caches. With 16 byte wide loads of an SSE register in an x86 CPU, it is a bad idea to use uncached loads since later accesses to the same cache line have to load the data from memory again (in case of cache misses). If, on the other hand, the vector registers are wide enough to hold one or more cache lines, uncached loads or stores do not have negative impacts. It becomes more practical to perform operations on data sets which do not fit into the caches.

Having large vector registers does not necessarily mean the latency of the instructions is increased; vector instructions do not have to wait until all data is read or stored. The vector units could start with the data which has already been read if it can recognize the code flow. That means, if, for instance, a vector register is to be loaded and then all vector elements multiplied by a scalar, the CPU could start the multiplication operation as soon as the first part of the vector has been loaded. It is just a matter of sophistication of the vector unit. What this shows is that, in theory, the vector registers can grow really wide, and that programs could potentially be designed today with this in mind. In practice, there are limitations imposed on the vector register size by the fact that the processors are used in multi-process and multi-thread OSes. As a result, the context switch times, which include storing and loading register values, is important.

With wider vector registers there is the problem that the input and output data of the operations cannot be sequentially laid out in memory. This might be because a matrix is sparse, a matrix is accessed by columns instead of rows, and many other factors. Vector units provide, for this case, ways to access memory in non-sequential patterns. A single vector load or store can be parametrized and instructed to load data from many different places in the address space. Using today's multi-media instructions, this is not possible at all. The values would have to be explicitly loaded one by one and then painstakingly combined into one vector register.

The vector units of the old days had different modes to allow the most useful access patterns:

- using *striding*, the program can specify how big the gap between two neighboring vector elements is. The gap between all elements must be the same but this would, for instance, easily allow to read the column of a matrix into a vector register in one instruction instead of one instruction per row.

- using indirection, arbitrary access patterns can be created. The load or store instruction would receive a pointer to an array which contains addresses or offsets of the real memory locations which have to be loaded.

It is unclear at this point whether we will see a revival of true vector operations in future versions of mainstream processors. Maybe this work will be relegated to co-processors. In any case, should we get access to vector operations, it is all the more important to correctly organize the code performing such operations. The code should be self-contained and replaceable, and the interface should be general enough to efficiently apply vector operations. For instance, interfaces should allow adding entire matrixes instead of operating on rows, columns, or even groups of elements. The larger the building blocks, the better the chance of using vector operations.

In [vectorops] the authors make a passionate plea for the revival of vector operations. They point out many advantages and try to debunk various myths. They paint an overly simplistic image, though. As mentioned above, large register sets mean high context switch times, which have to be avoided in general purpose OSes. See the problems of the IA-64 processor when it comes to context switch-intensive operations. The long execution time for vector operations is also a problem if interrupts are involved. If an interrupt is raised, the processor must stop its current work and start working on handling the interrupt. After that, it must resume executing the interrupted code. It is generally a big problem to interrupt an instruction in the middle of the work; it is not impossible, but it is complicated. For long running instructions this has to happen, or the instructions must be implemented in a restartable fashion, since otherwise the interrupt reaction time is too high. The latter is not acceptable.

Vector units also were forgiving as far as alignment of the memory access is concerned, which shaped the algorithms which were developed. Some of today's processors (especially RISC processors) require strict alignment so the extension to full vector operations is not trivial. There are big potential upsides to having vector operations, especially when striding and indirection are supported, so that we can hope to see this functionality in the future.

Appendices and bibliography

The appendices and bibliography page contains, among other things, the source code for a number of the benchmark programs used for this document, more information on oprofile, some discussion of memory types, an introduction to libNUMA, and the bibliography.

Index entries for this article
GuestArticles Drepper, Ulrich

(Log in to post comments)

Transactional cache size

Posted Nov 14, 2007 22:12 UTC (Wed) by **i3839** (guest, #31386) [Link]

It seems that the transactional cache should use a smaller cache size than normal caches, around 16 bytes or so.

Reply to this comment

latency can be hidden with enough parallelism

Posted Nov 15, 2007 0:37 UTC (Thu) by **stevenj** (guest, #421) [Link]

Another perspective that several people have advocated (e.g. Burton Smith), is that in the future if we can have enough fine-grained parallelism, then latency can be hidden. (e.g. you do one instruction from thread 1, the one instruction from thread 2, then... by the time you get back to thread 1, it is many cycles later and any memory operation will have completed). This was the underlying idea behind things like Intel's Hyper-threading stuff (which does it on a small scale) or the Tera MTA supercomputer (on a large scale).

One problem, of course, is extracting that much parallelism from most programs is still rather hard. There are very few programs out there that are ready to use 100 or 1000 processors.

(Parallelism is actually a huge problem on the computer-science horizon. Processors aren't getting much faster any more, and the only way to get increased performance is by exploiting parallelism, but writing parallel programs is still the province of an elite few.)

Reply to this comment

latency can be hidden with enough parallelism

Posted Nov 15, 2007 2:52 UTC (Thu) by **elanthis** (guest, #6227) [Link]

I imagine you could get quite a huge performance boost out of many applications by just writing them to not be so inefficient. That being one of the primary purposes of this series of articles. A lot of modern software blows through cycles completely needlessly. This has gone on because of the increase in processor speed - software could expand to fill the available increase in speed and reduce the complexity of the software in the process. Now that the choice is between the complexity of efficient software designs or the complexity of intense parallelism, maybe we'll see a shift back towards efficient code instead of quickly-written code.

Then again, probably not - it seems like it might end up being easier to make simple tools for parallel code than educating developers to write efficient code. :/

Call me crazy, but I take pride in being able to write single-threaded daemons that outperform heavily threaded implementations for even moderately high workloads. :p

Reply to this comment

latency can be hidden with enough parallelism

Posted Nov 20, 2007 2:42 UTC (Tue) by **nagual_sorcerer** (guest, #49143) [Link]

You are not crazy.:) But when there are multiple cpu cores, you still write single threaded daemons? Or just one thread per cpu core? I don't know if that goes faster, or there is some other way of doing things faster?

Reply to this comment

Section 9.1 (appendix)

Posted Nov 15, 2007 16:33 UTC (Thu) by **southey** (subscriber, #9466) [Link]

In '8.4 Vector Operations' you refer to '[t]he example in Section 9.1'.

Can you please make it clear that this Section is in the Appendix?

Also the Appendix refers to 'matrix multiplication in section Section 6.2.1' which probably should be Section 8.2.1.

Reply to this comment

Section 9.1 (appendix)

Posted Nov 20, 2007 22:58 UTC (Tue) by **cdarroch** (subscriber, #26812) [Link]

Section 6.2.1 is correct -- see http://lwn.net/Articles/255364/ where he writes, "the program code is pushed out into Section 9.1".

Reply to this comment

Memory part 8: Future technologies: transactional memory

Posted Nov 21, 2007 23:17 UTC (Wed) by **PaulMcKenney** (subscriber, #9624) [Link]

Transactional memory is indeed a very interesting technology, and there has been quite a bit of work in this area of late! Some researchers believe that they have solved the multiple-cache problem, for example. However, there are a few remaining challenges:

1. Non-idempotent operations (such as I/O) must be excluded from transactions.
2. Transactions can be subject to priority inversion, see for example Rossbach et al's SOSP paper.
3. Although there are proposals to handle very large transactions, these have performance consequences. So the performance of transactional memory usually depends heavily on the size and associativity of the hardware caches.

4. Resolving contention among transactions turns out to be non-trivial, though of course designing for low contention is the best approach, and not just for transactional memory!
5. As far as I know, transactional memory implementations do not yet take realtime issues into account.
6. Efficient handling of read-mostly situations is best served by allowing concurrent transactional and non-transactional access to the same locations, which causes extreme discomfort to most of the transactional-memory people I have talked with.

All that aside, transactional memory certainly does seem like a much better approach than non-blocking synchronization!

Reply to this comment

Section 8.3

Posted Nov 22, 2007 22:38 UTC (Thu) by **anton** (subscriber, #25547) [Link]

FB-DRAM has already been deployed for quite some time in current multi-socket Xeon systems, and IIRC also in Niagara-based Suns. However, the high power consumption and heat dissipation of FB-DIMMs leads Intel to also provide Dual-Xeon chipsets that support plain registered DIMMs; unfortunately, the higher pin count of the regular DDR2 interface mean that this chipset supports only two channels instead of four. AMD still has an advantage here (two DDR2 channels per socket).

Concerning Cell SPUs and memory latency: the SPUs can only access their local memory (256KB), not the main memory. If data or code needs to be loaded into the local memory, software has to program the transfer from main memory explicitly (unlike cache, which is automatic). So, yes, explicit prefetching is necessary, otherwise the SPU won't get it's data at all.

Reply to this comment