I think we should back up here and ask ourselves a couple of
questions:

1. What are we trying to accomplish here?

2. Is this the best way to accomplish it?

To the first question, the problem as I understand it as follows:
Heavyweight locks don't conflict between members of a parallel
group.
However, this is wrong for LOCKTAG_RELATION_EXTENSION, LOCKTAG_PAGE,
LOCKTAG_TUPLE, and LOCKTAG_SPECULATIVE_TOKEN. Currently, those
cases
don't arise, because parallel operations are strictly read-only
(except for inserts by the leader into a just-created table, when
only
one member of the group can be taking the lock anyway). However,
once
we allow writes, they become possible, so some solution is needed.

To the second question, there are a couple of ways we could fix
this.
First, we could continue to allow these locks to be taken in the
heavyweight lock manager, but make them conflict even between
members
of the same lock group. This is, however, complicated. A
significant
problem (or so I think) is that the deadlock detector logic, which
is
already quite hard to test, will become even more complicated, since
wait edges between members of a lock group need to exist at some
times
and not other times. Moreover, to the best of my knowledge, the
increased complexity would have no benefit, because it doesn't look
to
me like we ever take any other heavyweight lock while holding one of
these four kinds of locks. Therefore, no deadlock can occur: if
we're
waiting for one of these locks, the process that holds it is not
waiting for any other heavyweight lock. This gives rise to a second
idea: move these locks out of the heavyweight lock manager and
handle
them with separate code that does not have deadlock detection and
doesn't need as many lock modes. I think that idea is basically
sound, although it's possibly not the only sound idea.

However, that makes me wonder whether we shouldn't be a bit more
aggressive with this patch: why JUST relation extension locks? Why
not all four types of locks listed above? Actually, tuple locks are
a
bit sticky, because they have four lock modes. The other three
kinds
are very similar -- all you can do is "take it" (implicitly, in
exclusive mode), "try to take it" (again, implicitly, in exclusive

mode), or "wait for it to be released" (i.e. share lock and then release). Another idea is to try to handle those three types and leave the tuple locking problem for another day.

I suggest that a good thing to do more or less immediately, regardless
of when this patch ends up being ready, would be to insert an insertion that LockAcquire() is never called while holding a lock of one of these types. If that assertion ever fails, then the whole theory that these lock types don't need deadlock detection is wrong, and we'd like to find out about that sooner or later.

On the details of the patch, it appears that RelExtLockAcquire() executes the wait-for-lock code with the partition lock held, and then
continues to hold the partition lock for the entire time that the relation extension lock is held. That not only makes all code that runs while holding the lock non-interruptible but makes a lot of the rest of this code pointless. How is any of this atomics code going to
be reached by more than one process at the same time if the entire bucket is exclusive-locked? I would guess that the concurrency is not
very good here for the same reason. Of course, just releasing the bucket lock wouldn't be right either, because then ext_lock might go away while we've got a pointer to it, which wouldn't be good. I think
you could make this work if each lock had both a locker count and a pin count, and the object can only be removed when the pin_count is 0.
So the lock algorithm would look like this:

- Acquire the partition LWLock.
- Find the item of interest, creating it if necessary. If out of memory for more elements, sweep through the table and reclaim 0-pin-count entries, then retry.
- Increment the pin count.
- Attempt to acquire the lock atomically; if we succeed, release the partition lock and return.
- If this was a conditional-acquire, then decrement the pin count, release the partition lock and return.
- Release the partition lock.
- Sleep on the condition variable until we manage to atomically acquire the lock.

The unlock algorithm would just decrement the pin count and, if the resulting value is non-zero, broadcast on the condition variable.

Although I think this will work, I'm not sure this is actually a great
algorithm. Every lock acquisition has to take and release the partition lock, use at least two more atomic ops (to take the pin and
the lock), and search a hash table. I don't think that's going to

be
staggeringly fast. Maybe it's OK. It's not that much worse,
possibly
not any worse, than what the main lock manager does now. However,
especially if we implement a solution specific to relation locks, it
seems like it would be better if we could somehow optimize based on
the facts that (1) many relation locks will not conflict and (2)
it's
very common for the same backend to take and release the same
extension lock over and over again. I don't have a specific
proposal
right now.

Whatever we end up with, I think we should write some kind of a test
harness to benchmark the number of acquire/release cycles per second
that we can do with the current relation extension lock system vs.
the
proposed new system. Ideally, we'd be faster, since we're proposing
a
more specialized mechanism. But at least we should not be slower.
pgbench isn't a good test because the relation extension lock will
barely be taken let alone contended; we need to check something like
parallel copies into the same table to see any effect.