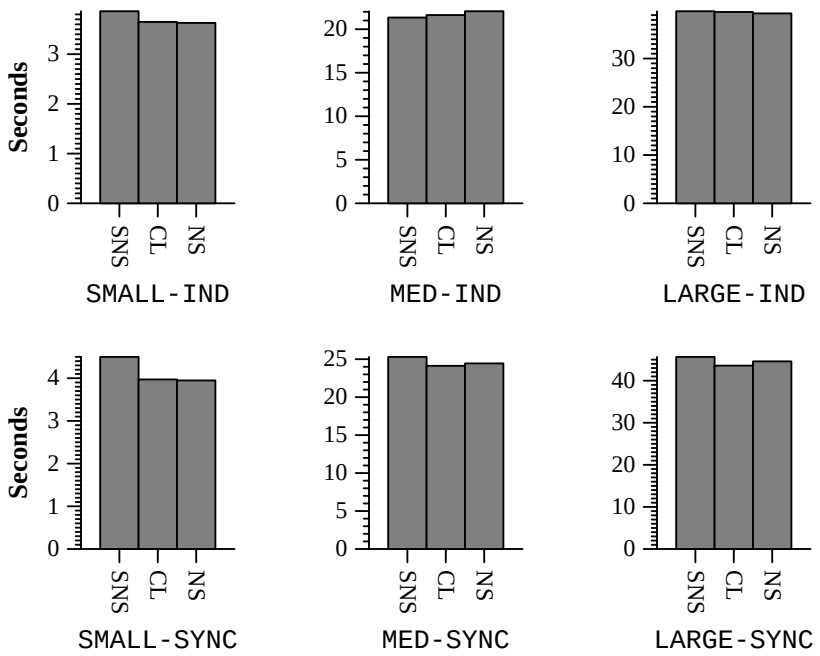

Figure 4.7: Checkpoint Overhead of Basic Algorithms.

there are no m_{CL} messages sent in the staggering version of the NS algorithm. Thus, the degree of staggering is much greater than in the `SYNC` instances.

The staggering causes predictable results in Figure 4.9 as well. In the `SYNC` instances of `a_m_s`, staggering increases the overhead of the algorithm. This is because the processors spend extra time waiting to receive synchronization messages from processors that are busy checkpointing, as illustrated in Figure 3.7 of Chapter 3. In the `IND` algorithms, however, staggering is beneficial to checkpoint overhead, since there are no synchronization messages upon which processors may stall, and the staggering helps reduce contention on the disks. It may be recalled that there are four disks that CFS uses to perform file writes. Moreover, CFS stripes files on those four disks. Thus, even though CFS reduces the overhead of file I/O from file systems that go through the SRM, the disks are still a source of contention: Staggering the checkpoints is indeed a good way to reduce the overhead of checkpointing. As further proof, when the `LARGE-IND` instance of `a_m_s` was checkpointed with the non-staggered version of the NS algorithm, each processor took an average

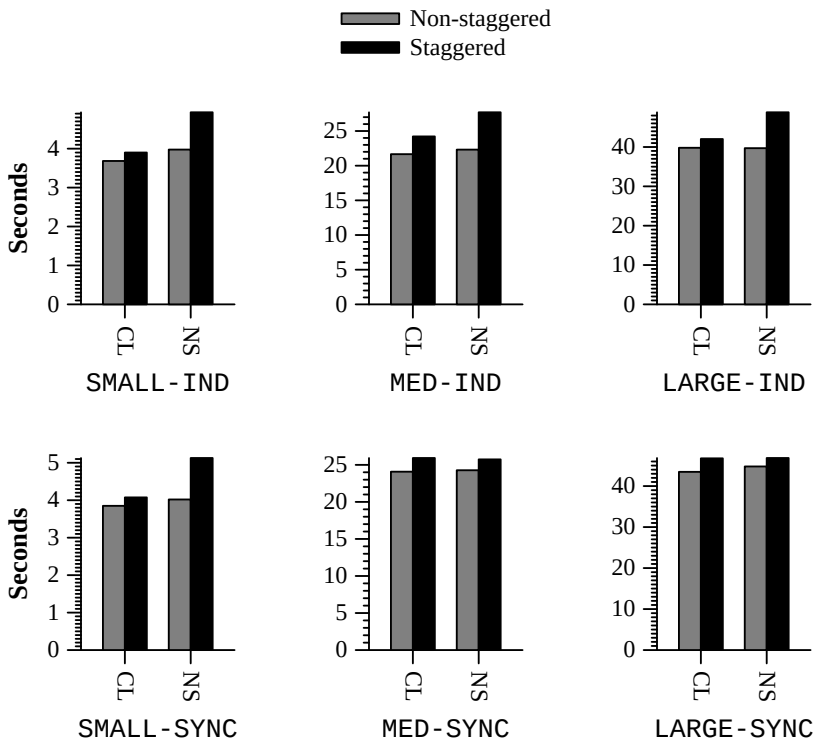

Figure 4.8: Effect of Staggering on Checkpoint Time.

of 34 seconds to write its checkpoint to disk. In the staggered version, the average was 12 seconds.

4.2.6 Main Memory Checkpointing

Figures 4.10 and 4.11 show the results of using the main memory optimization to improve checkpoint overhead. The non-staggering versions of the CL and NS algorithms were used. Moreover, each node of the iPSC/860 was limited to four megabytes of physical memory. The reason for this is to make the `LARGE-IND` and `LARGE-SYNC` instances of `a_m_s` have address spaces that are too large to copy into main memory. When such a situation occurs, the checkpointer writes enough of the checkpoint to disk while the processors are frozen that the rest of the checkpoint fits into main memory. This last fragment of the checkpoint

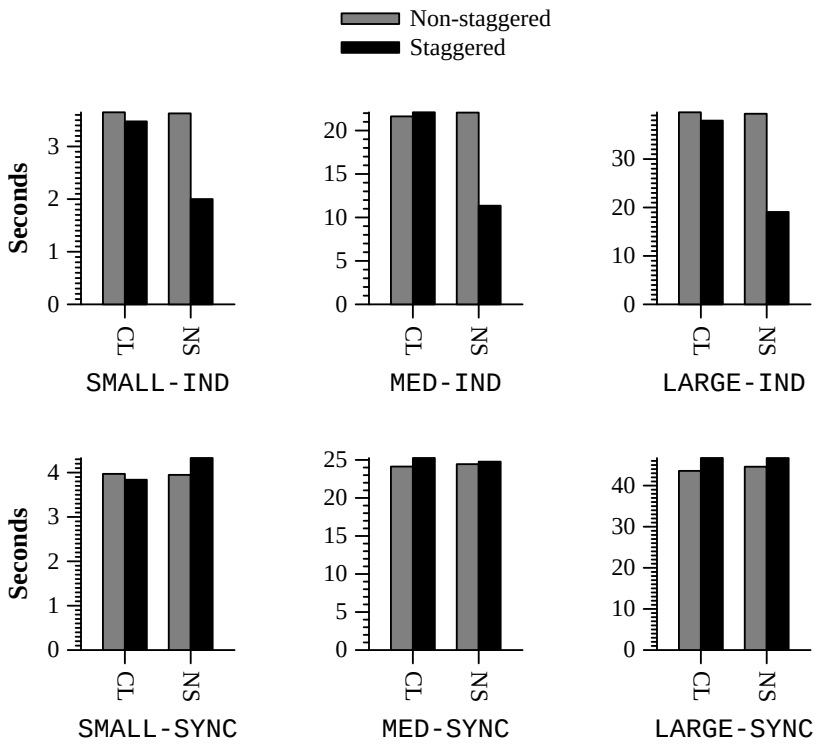

Figure 4.9: Effect of Staggering on Checkpoint Overhead.

is then copied to main memory, and the processors are restarted before writing the fragment to disk.

As mentioned in Section 3.4, main-memory checkpointing is extra beneficial in multicomputer checkpointing, for when the main memory checkpoint fits into physical memory, the checkpointer may write the checkpoint to disk at its leisure. Thus, ICKP attempts to reduce disk contention by enforcing serialization among the processors' disk writes. That is, processor 0 writes its checkpoint to disk immediately after restarting its target program. Thereafter, processor n must wait for processor $n-1$ to finish writing its checkpoint to disk before it may write its checkpoint.

Figure 4.10 shows how these optimizations affect the overall checkpoint time. For the `SMALL` and `MED` instances of `a_m_s`, the main memory algorithms increase the checkpoint time significantly. This is because of the extra time copying bytes, and because of the extra time that processors spend waiting

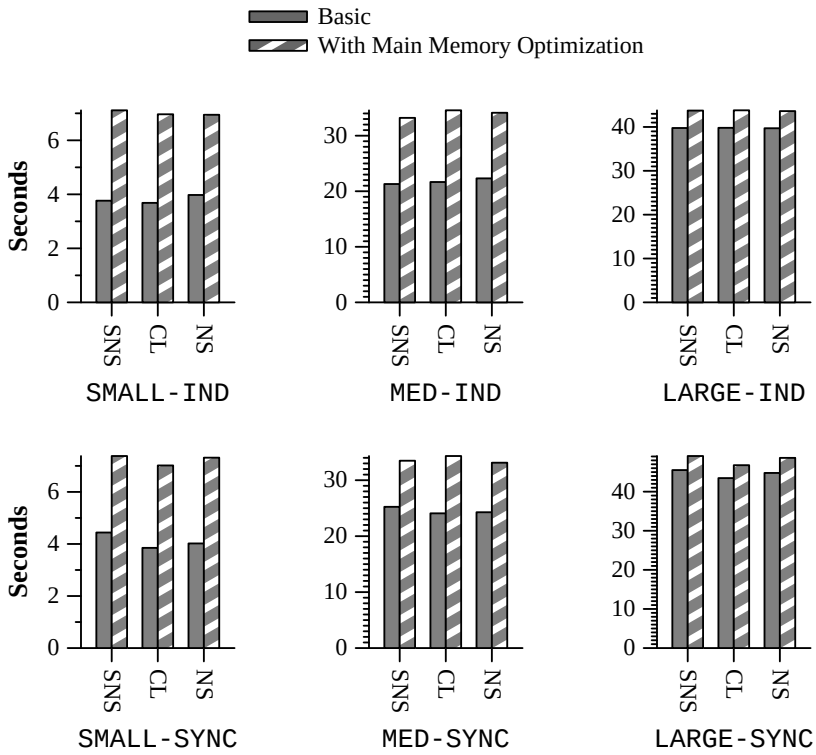

Figure 4.10: Effect of Main Memory Checkpointing on Checkpoint Time.

for their turn to write their checkpoint to disk. However, Figure 4.11 shows that the optimizations have a dramatically beneficial effect on the checkpoint overhead. Essentially, the only overhead is the memory-to-memory copy time.

In the `LARGE` instances, the increase of the checkpoint time and the decrease of the checkpoint overhead are not nearly so dramatic. This is because the main memory checkpoint does not fit into physical memory, and thus, most of it has to be written to disk while the processors are frozen. Moreover, since the checkpoint does not fit into main memory, the disk writes cannot be staggered and thus contention cannot be reduced in that manner. Still, this way of dealing with the limitations of physical memory is far superior than in the Firefly checkpointer, where the main memory optimization causes thrashing in the virtual memory system if the main memory checkpoint does not fit. Here, the main memory optimization still improves checkpoint overhead, albeit to a much smaller degree than when the checkpoint fits into physical memory.

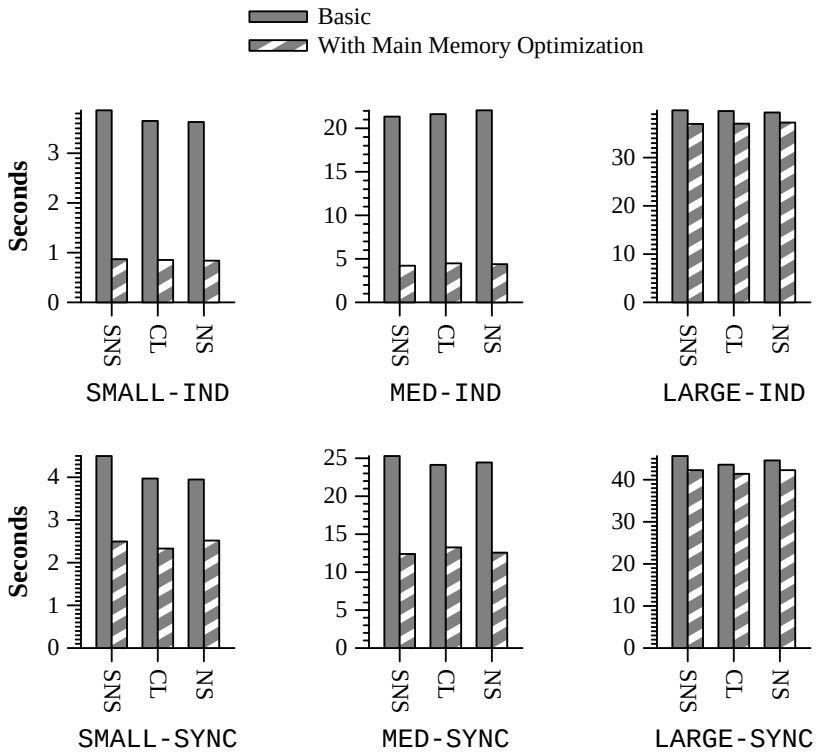

Figure 4.11: Effect of Main Memory Checkpointing on Checkpoint Overhead.

Figures 4.12 and 4.13 show the effects of staggering the main memory algorithms. As expected, in the `SMALL` and `MED` instance of `a_m_s`, there is little change between staggering and non-staggering. This is because both main memory algorithms can stagger the writes to disk once they restart the processors. In the `LARGE` instances, there is little change from the results of the basic algorithms. This is because most of the checkpoint is written to disk while the processors are frozen, and therefore staggering helps in the `LARGE-IND` case, and hurts in the `LARGE-SYNC` case.

The checkpoint overheads measured with the `MED-SYNC` instance of `a_m_s` increases with staggering for the same reason the overhead increases in Figure 4.9: Because of the staggering, the processors have to wait an extra period of time to receive their synchronization messages.

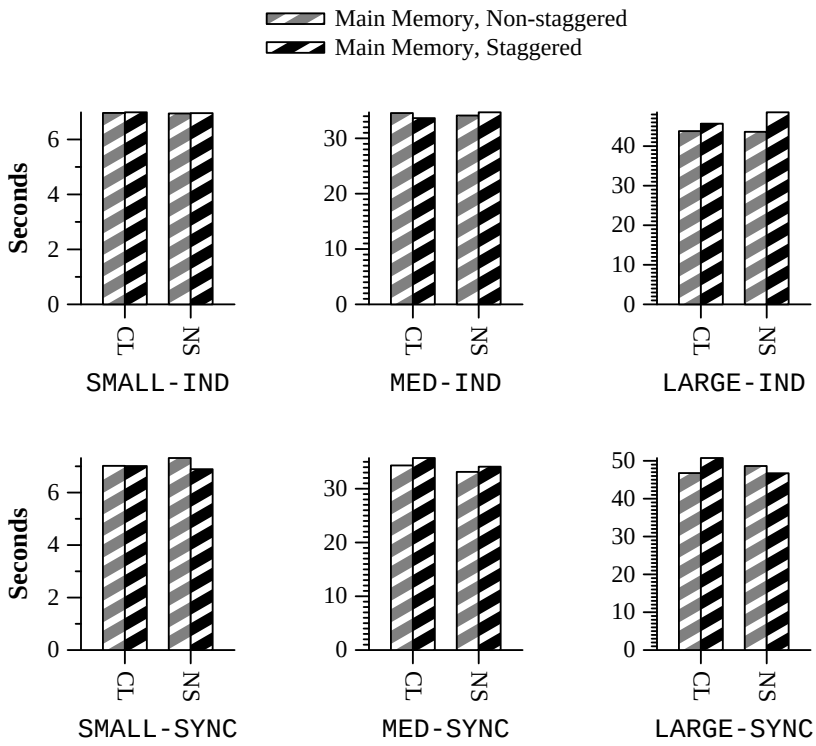

Figure 4.12: Effect of Staggering on Main Memory Checkpoint Times.

4.2.7 Results of Compression

The compression algorithm implemented in ICKP is a slight variation of an algorithm defined by Wheeler and called "Algorithm 1" in [BJLM92]. It was chosen as the algorithm of choice for on-line data compression in their log-structured file system, because it achieves decent compression ratios for most data files, and is much faster than more traditional compression algorithms, such as the "LZC" algorithm used in the Unix `compress` utility [BWC89]. The algorithm works as follows: While compressing, maintain a hash table that hashes on the last three bytes compressed. If the next byte matches the value stored in the hash table, then emit a '0'. Otherwise, emit a '1' followed by the byte, and update the value in the hash table.

ICKP implements a variation of the algorithm that adds run-length encoding: Instead of emitting a '0' on a hash table hit, simply increment a

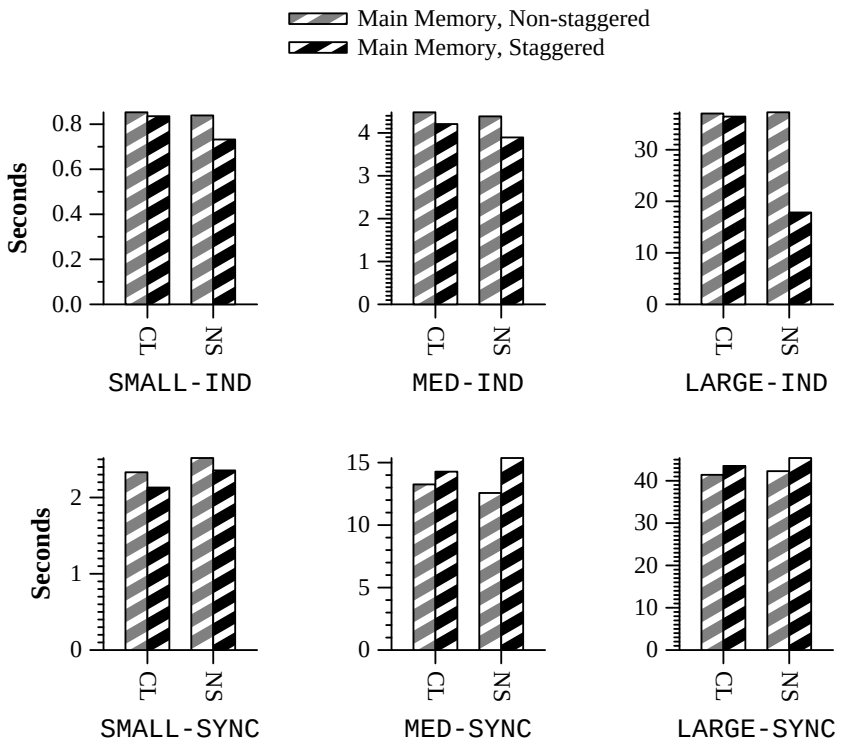

Figure 4.13: Effect of Staggering on Main Memory Checkpoint Overhead.

counter called *nhits*. Upon a miss, if *nhits* is greater than zero, then emit $\lfloor \log_2 nhits \rfloor + 1$ zeros, followed by *nhits* starting with its most significant bit. Then emit a '1' and reset *nhits* to zero. This variation improves the compression ratio of the algorithm when the hash table hits come in groups. For example, a sequence of 128 hash table hits will yield 16 bytes of zeros in the original algorithm. The same sequence will yield just two bytes with run-length encoding: `'00000000 1000000'`. When the hash hits are sparse, then the variation can degrade compression by up to one bit per hit.

We chose the variation for ICKP because address spaces often contain large chunks of uninitialized data. As will be shown later, this is especially true in many FORTRAN programs, as FORTRAN requires static allocation of memory, and thus many programmers preallocate huge chunks of memory in their programs so that they can scale up to larger data sizes without recompilation.

Moreover, in the case where the variation degrades compression, hash table hits are sparse, and thus the compression factor will not be very large.

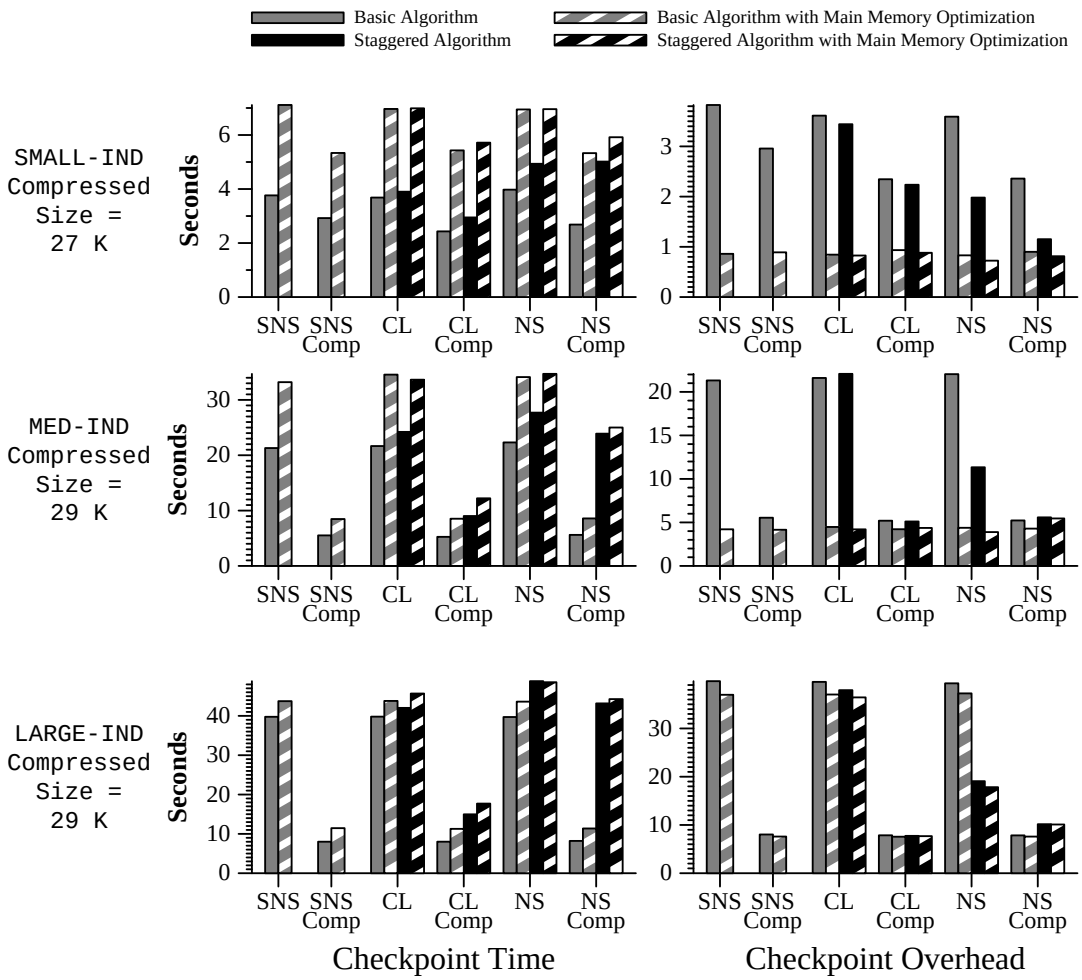

Figure 4.14: Effect of Compression on the `IND` Instances of `a_m_s`

Figures 4.14 and 4.15 show the results of compression on every consistent checkpointing algorithm and optimization. Note that since the `MED` and `LARGE` instance of `a_m_s` allocate large chunks of zeros, their checkpoints are compressed down to very small sizes. Thus in these instances, compression improves almost every algorithm in both checkpoint time and overhead. Note that the compressed and staggered optimizations of the Network Sweep algorithm have a drastic increase in checkpoint time over the non-staggered

versions. This is because the time to compress the checkpoint is larger than the time to make a memory-to-memory or disk copy, and thus the degree of staggering is larger than when the checkpoints are not compressed. This degree of staggering is responsible for the drastic increase in checkpoint overhead for the SYNC instances of a_m_s: Staggering is heavily penalized because the processors have to wait extra long for synchronization messages.

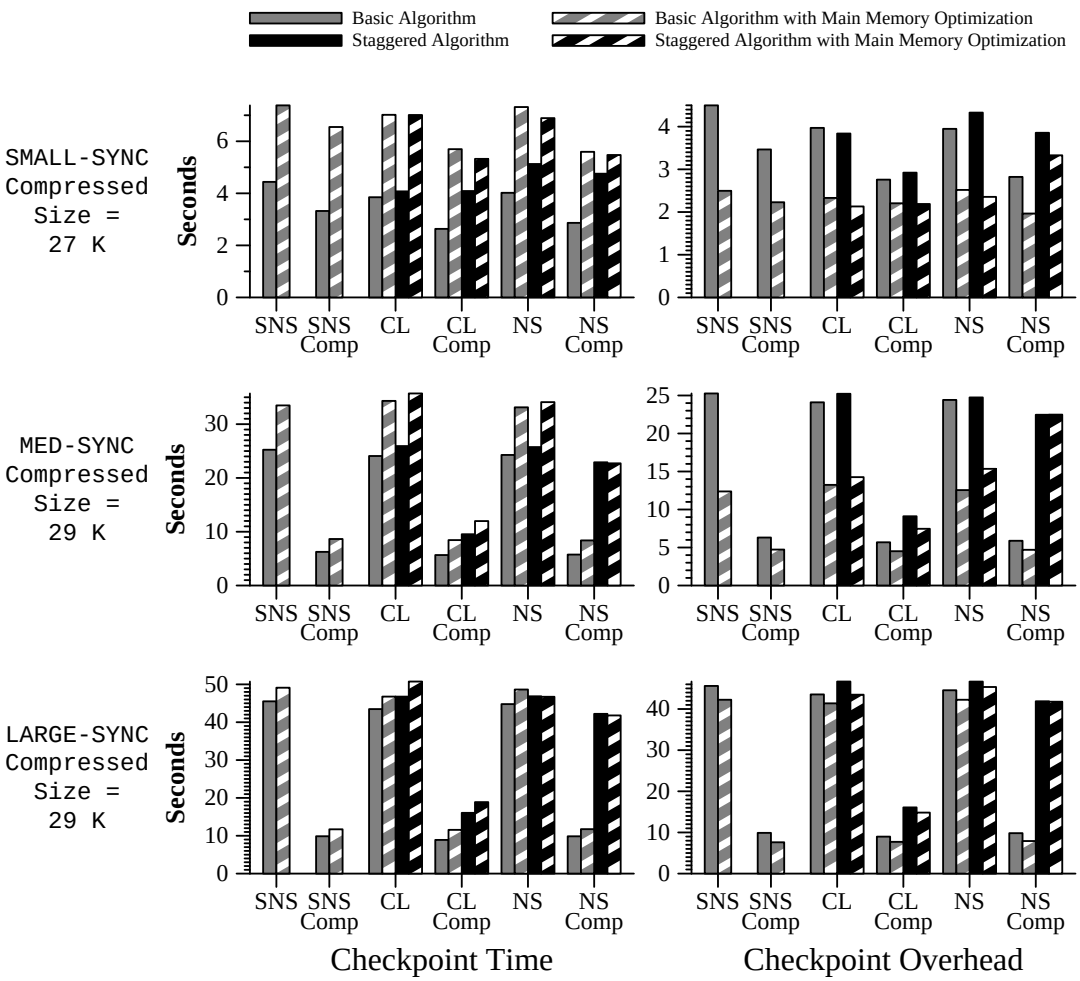

Figure 4.15: Effect of Compression on the SYNC Instances of a_m_s

The compression of the SMALL instances of a_m_s are interesting. One would expect that since the program allocates almost no extra memory, the address space would not compress very well. However, it is compressed by

more than 84%. Thus, its checkpointing times are reduced with compression. Yet, since the checkpoints files are so small, the checkpoint overheads of the main memory optimizations are not improved. This is because the increase in the time that it takes to do the compression is not offset by the reduction in time that it takes to do the write.

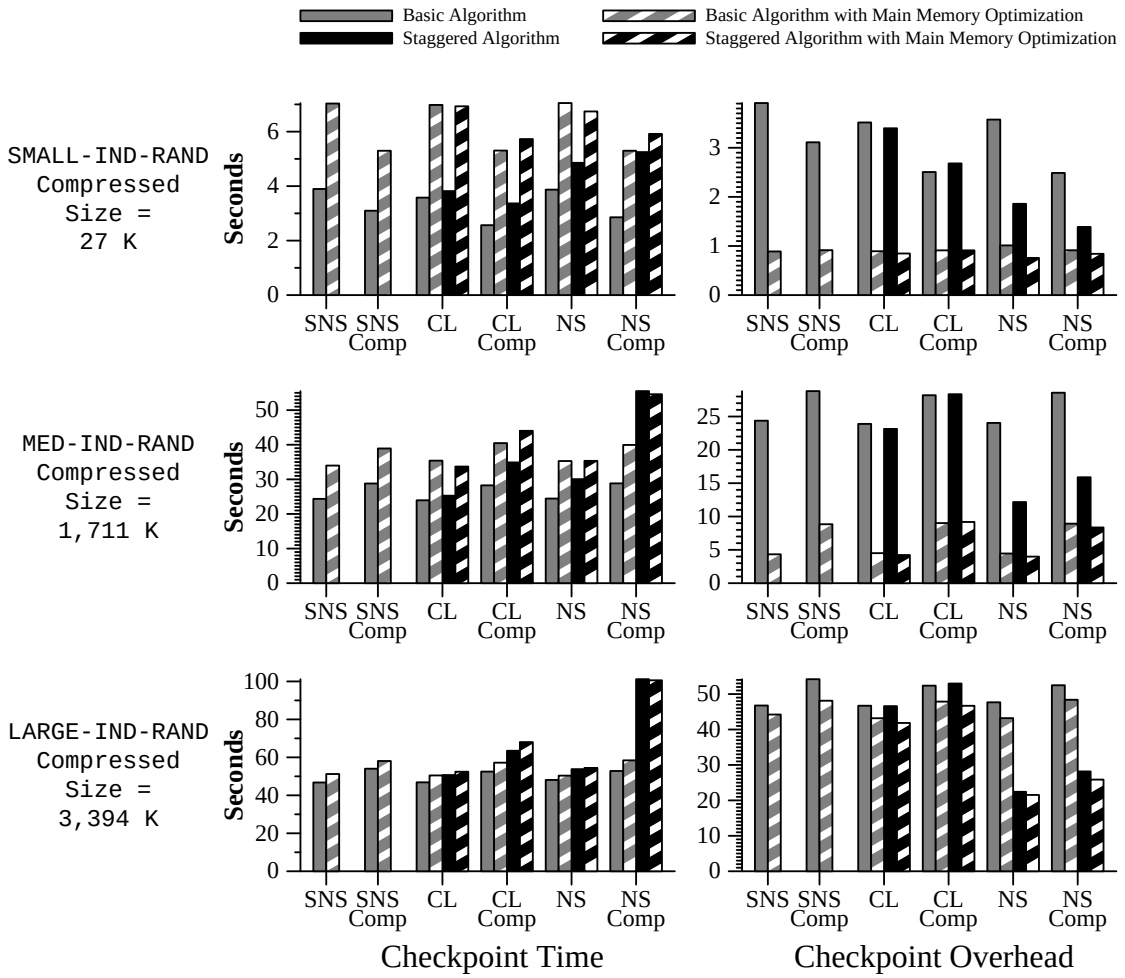

Figure 4.16: Effect of Compression on Random `a_m_s` Programs

The graphs in Figure 4.16 show the opposite extreme of compression: The bytes that `a_m_s` allocates are set randomly, and the compression algorithm fails to compress the checkpoint. Thus, compression is doubly bad—it increases the size of the checkpoint, and the compression takes longer than a