

mechanism (at the cost of requiring additional memory space) or even turned off (at the cost of being able to recover only from some transient errors).

## 2.8 Evaluation

To evaluate ReVive, we examine three issues: overhead in error-free execution, storage requirements, and recovery overhead.

### 2.8.1 Overhead in Error-Free Execution

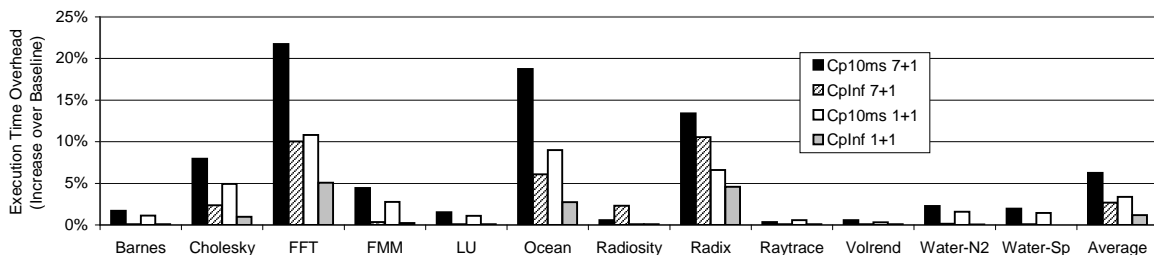


Figure 2.9: Performance overhead of ReVive in error-free execution.

To evaluate the impact of ReVive on error-free execution, we compare ReVive to a baseline system that includes no recovery support. As explained in Section 2.5.1, the sources of performance overhead in error-free execution with ReVive are parity and log updates, and checkpoint generation. For given cache sizes and other machine parameters, the overhead of parity and log updates mainly depends on the characteristics of the application being executed. The overhead of checkpoint generation depends on the frequency of checkpointing. To better understand these overheads, Figure 2.9 shows the performance overhead of our mechanism using 7+1 parity and with checkpoints performed every 10ms (**Cp10ms**) and with an infinite checkpoint interval (**CpInf**). For comparison, we also show the results of our scheme when mirroring is used instead of parity (as described in Section 2.4.1), for the same checkpoint frequencies: once every 10ms (**Cp10msM**) and with an infinite checkpoint interval (**CpInfM**). The **CpInf** and **CpInfM** bars reveal the overheads of logging and parity maintenance with 7+1 parity and mirroring, respectively. The difference between **Cp10ms** and **CpInf**, and between **Cp10msM** and **CpInfM**, represents the overhead of establishing checkpoints every 10ms, using 7 + 1 parity and mirroring, respectively.

The average overhead of logging and parity maintenance is low, 2.7% for 7+1 parity (CpInf) and 1% for mirroring (CpInfM). In applications with important working sets that do not fit in the L2 cache (FFT, Ocean, and Radix), this overhead can be high. It reaches 11% in Radix.

The overhead of establishing checkpoints every 10ms is usually small, but it can be relatively high, as in FFT and Ocean. When the checkpoint is established in these applications, almost all lines in their caches are dirty, so the checkpoint takes close to worst case time. In FFT, this effect combines with the high logging and parity maintenance overheads for an overall overhead of 22%, the highest overhead we observe in any of the twelve applications. It is important to note that a checkpoint interval of 10ms is the least favorable end of the spectrum for our scheme. Increasing the checkpoint interval or simply using mirroring instead of parity can reduce the overhead to 10% in FFT. When mirroring is used and the checkpoints are infrequent, the overhead is reduced to 5% on FFT and 1% on the average.

ReVive can be designed to be configured at boot time to support parity or mirroring. If the machine is mostly going to run applications that exhibit good caching behavior, the performance overheads are small and parity should be used to reduce the memory space overhead (Section 2.8.2). For applications with poorer caching behavior, a tradeoff exists between memory space overheads and performance: mirroring is faster but uses more memory. In reality, parity and mirroring need not be used in a mutually exclusive fashion. For example, a small part of the memory can be protected by mirroring, while the rest is protected by parity. Careful allocation of frequently used pages into the mirrored region should result in low overheads, as most of the memory modifications result in mirroring updates, while reducing the memory space overheads, as most of the memory space is uses the efficient parity approach.

To help understand the overheads observed, Figures 2.10 and 2.11 show the network and memory traffic in the machine with the Cp10ms configuration<sup>5</sup>. The breakdown of the traffic is as follows: RD/RDX represents the traffic due to supplying the data on cache misses; Exe WB is the traffic due to writing back dirty lines to memory in regular execution; Ckp WB is the traffic due to writing back

---

<sup>5</sup>Note that traffic shown in Figures 2.10 and 2.11 is substantially lower than that reported in [40] in Figures 9 and 10. The traffic reported in [40] is incorrect as result of a script bug. However, the error substantially changes only the scale of the vertical axis in each of the figures, and none of the conclusions in [40] depend on that. We have checked the other results reported in [40] and found that the error was confined to Figures 9 and 10. We thank Prof. Marc Snir for calling our attention to this problem.

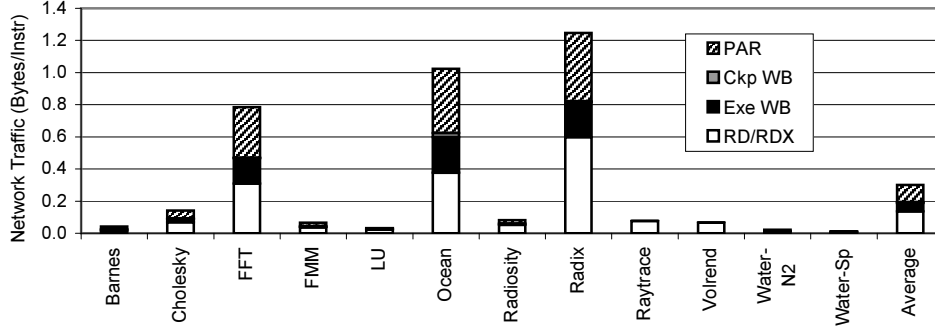


Figure 2.10: Breakdown of network traffic<sup>5</sup> in Cp10ms.

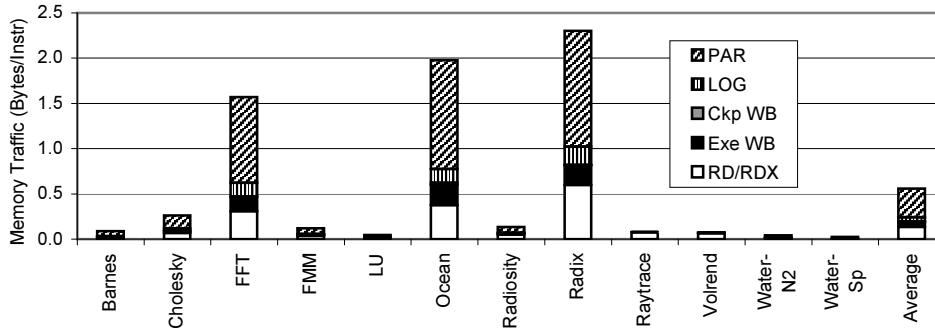


Figure 2.11: Breakdown of memory traffic<sup>5</sup> in Cp10ms.

dirty lines when checkpoints are established; LOG is the traffic of writing data to the logs; PAR is the traffic due to parity updates (for both data and logs). Traffic shown as RD/RDX and Exe WB is the same as in the baseline system. Traffic shown as Ckp WB, LOG, and PAR is caused by ReVive. If mirroring was used instead of parity, the network traffic would stay the same as in Figure 2.10; the memory traffic would change only in that PAR would shrink to one-third of its size.

Figures 2.10 and 2.11 show that, for most of the applications, both the network and the memory traffic are low, without or with ReVive. The exceptions are FFT, Ocean, and Radix, where traffic is already high in the baseline system. For these three applications, the additional traffic, mostly resulting from parity maintenance, further degrades the already poor performance.

We note that, for systems and applications where performance is constrained by bandwidth, ReVive can result in significant overheads. For such systems designers should provide additional bandwidth to support ReVive - although this increases system cost. Furthermore, without ReVive systems are typically designed so that the available bandwidth between memory and the memory or

directory controller matches the bandwidth of the bus or interconnect. In ReVive, the bandwidth requirements for these two are different - the required bandwidth between the directory controller and its local memory is higher because of parity updates. Thus, when designing ReVive into a system that is expected to run applications with high memory bandwidth requirements, additional bandwidth between the local memory and its directory controller should be provided. Alternatively, the use of mirroring instead of parity should be considered in such systems.

## 2.8.2 Storage Requirements

ReVive requires additional memory space to store distributed parity and logs. Now we determine the memory space overheads caused by storing distributed parity for the ReVive configuration used in this evaluation. We also experimentally determine the memory space overheads that result from storing logs in main memory.

### Parity Storage Requirements

To keep the hardware simple, the number of nodes should be a multiple of the parity group size. In addition, the latter should be a power of two, so that to determine which node has the parity page for a given group, we can use a trivial implementation of the *mod* operation. With 7+1 parity, 88% of the main memory is used for data, while 12% is used for parity. We can reduce this requirement by employing larger parity groups. However, doing so slows down recovery and increases the risk of contention in the home of a parity page belonging to a particularly popular parity group. If mirroring is used instead of parity, the overhead is 50% of the memory.

### Log storage requirements

Figure 2.12 shows the maximum log size for different applications for the Cp10ms configuration, assuming that logs for two most recent checkpoints are kept. As we can see, the largest log is about 2.5MB. With the conservative assumption of a log growing proportionally to the checkpoint interval, that yields 25MB for a checkpoint interval of 100ms. In reality, we expect the actual size to be significantly less, as longer intervals allow more filtering out of redundant log entries (Section 2.4.2).