Chapter 10

Alternative applications for reversibility

Most of this thesis has focused on the benefits of reversibility that are gained through its reduction of the energy dissipation of computation. However, full reversible operation may be useful for other reasons as well. In this short chapter we briefly survey some of the possibilities.

10.1 Auditable/verifiable/trustable computation

One interesting application for full reversibility in a computer system is to assist in meeting requirements for auditable or verifiable computation. This requirement exists in at least two forms for which reversibility might be useful:

- It should be possible to determine, with high confidence, whether a transient hardware error of some sort (e.g., a random bit-flip in memory) might have occurred during a computation, to help us determine whether the result of the computation can be considered valid.
- The system should ensure that any malicious intruder not having physical access to the hardware is unable to destroy any information stored in the system, and that the intruder's presence and complete actions can always be determined after the fact.

Let us examine how reversibility might be useful for helping to satisfy these requirements.

10.1.1 Detecting transient errors

Full reversibility could be used to detect transient hardware errors as follows. The initial state for the computation is set up, and we record the entire initial state, or if that is too large, a cryptographically secure hash of the entire initial state. Then a computation is run, and produces some result. Then we reverse the processor direction, and run in reverse, back to the initial state. Then we compare the state with the stored state or the checksum. If there are any differences, then some error must have occurred during the computation, and the result should be considered untrustworthy.

If the processor is designed to guarantee reversibility at the hardware level, then any differences in initial state will indicate that a hardware error occurred. However, if the processor only guarantees reversibility under an assumption of the correctness of some piece of software, then differences in initial state could indicate the presence of either a hardware error or an error in that software.

Hardware errors. If the error is a transient hardware error due to some random influence from the environment (such as a cosmic ray shower, perhaps), then it is very unlikely that an error would be missed by this detection scheme, as this would require two independent error events, one during the forward computation and one during the reverse computation, that happened to cancel each other out so that the identical initial state was reached even though the final result might have been corrupted.

If such a transient hardware error is detected, then the user can simply try running the computation again, and repeat until the computation proceeds forwards and backwards with no error being detected. In this way, a very high-confidence result can always be obtained eventually, even if there is a significant probability of transient hardware errors occurring during an individual run of the computation.

Of course, an alternative technique for detecting and correcting transient errors, without resorting to reversibility, is to run multiple copies of a system side-by-side, or run a single machine multiple times, and compare the results, and perhaps compare the entire state at checkpoints along the way.

Reversibility may also be useful for detecting the presence of some kinds of permanent hardware faults, since if the fault is permanent, then presumably the difference between initial states would be the same each time. The time at which the fault first makes itself felt could be rapidly determined by running forwards and backwards for different lengths of time in a binary search pattern. Note, however, that some permanent faults, such as a bitwise logical-complement instruction that consistently fails to flip a certain bit (in both the forwards and reverse directions), may maintain reversibility, and so will not be detected by the process of comparing initial states.

Note also that we will not necessarily catch transient or permanent hardware errors if they affect the state-comparison process as well as (or instead of) the main computation.

Software errors. If the system does not guarantee reversibility at the hardware level, the above technique will also catch those software errors that happen to lead to forward computations that fail to match the corresponding reverse computations. However, other kinds of software errors will not be caught. For example, if the system compiler correctly guarantees that all programs will be correctly reversible, then any errors in user programs compiled by that compiler will remain undetected by this technique.

Note also that if we detect an error in reversibility that is a software error, then an easy way to pinpoint its precise location is to run partial computations forwards and backwards, and find exactly how far forwards you must go before the reversibility of the system is corrupted. A binary search can be used to pinpoint the precise time of the error after O(log(n)) partial computations, where n is the number of steps in the entire computation. The machine state at this time can be examined to help determine the cause of the error. This technique will be a useful tool in debugging software that is intended to ensure reversibility on hardware that does not by itself guarantee reversibility. However, this will of course incur the usual reversibility cost of slower execution time, on traditional serial processors.

10.1.2 Logging or limiting effects of unwelcome intrusions

Suppose we have a requirement that any computer cracker that manages to get past system security measures should (1) be unable to actually permanently destroy any data, and (2) have the complete history of his actions on the system be determinable once his interference is discovered.

A traditional approach to item 1 is to make backups of data and log user activities, but this does not help if the cracker destroys data before it manages to get backed up, or corrupts the backup software itself so that new data does not get backed up properly.

However, if the system guarantees full reversibility at some level that is impossible for the cracker to interfere with (e.g., at the hardware level), then by the definition of reversibility, whatever actions he takes cannot permanently destroy any user data, or any information that the cracker had input to the system in order to do his dirty work.

Additionally, once the presence of the cracker is detected, the machine can be disconnected from the network and reversed to recover any desired earlier state of execution, and all of the cracker's actions can be observed, and the clean state prior to his break-in can be recovered.

However, note that reversibility does not protect us from the cracker's corrupting the system's outputs after the time he breaks in. He could in general still alter the running state of the system so that it produced invalid or misleading data until his interference is discovered, and its effects on the system are undone. Any inputs to the system while it was in a corrupted state would have to be re-input once a clean state is restored. All those inputs could be recovered by backing up over the time after the cracker's interference.

But this suggests an alternative technique for achieving the same protection without requiring reversibility. Namely, one could simply have an incorruptible mechanism for recording the initial state of the system (when it is first turned on in a clean-slate state) and for recording every bit of information that flows into the system, including any timing information, if that is important. If the system is deterministic, then that stored information is sufficient for reconstructing the complete machine state at any later time—the system is "reversible" in the sense that we could always back up to the state at any earlier time by simply going back to the initial state and proceeding forwards from there.

The obvious drawback to this technique is that if the system has been running continuously for a long time, say a year, and we only want to back up a small amount, we will have to spend another year to get up to the desired point. But then, the obvious solution is to also have an incorruptible mechanism for checkpointing the system state periodically, so that we can just go forwards from the last checkpoint.

In summary, although reversibility may be useful for tracing the activity of malicious crackers, and preventing them from damaging any data, it is not theoretically any better than just reliably recording the system's initial state and all inputs. It may or may not turn out to be easier to implement. Thus, this application is not, by itself, a convincing justification for reversibility.

10.2 Program debugging

One interesting application of a reversible instruction set is that it makes it very simple to write a bi-directional debugger, which allows stepping backwards as well as forwards through a program. This feature eases the software debugging process, since, when observing that the program is behaving incorrectly, one can simply run in reverse from the point where the problem was first observed, to quickly trace back through the preceding events that led to the errant behavior.

Our simulator for the Pendulum instruction set (written by Matt DeBergalis) has the feature that one can step backwards as well as forwards through the program code, while observing registers. The simulator is thus a simple example of a bi-directional debugger, at the level of assembly instructions.

During the development of the compiler discussed in §9.4.3, these bi-directional debugging capabilities proved very useful several times, for tracking down the causes of incorrect program behavior caused by bugs in the compiler. During the compiler development process, we were using a version of the Pendulum instruction set that guaranteed reversibility independently of program well-formedness. This allowed the bidirectional capabilities of the simulation/debugging envrionment to function even when the compiler still had bugs. Incorrect program behavior was tracked backwards in time until the instructions that had caused the inappropriate behavior were found, at which point the compiler could easily be fixed.

So we have seen that a reversible computing capability can ease debugging. However, reversible computing is not strictly necessary for implementing a bi-directional debugger. For example, Boothe (1998, [27]) describes algorithms that can be used to implement bi-directional debugging environments for normal (irreversible) programming languages. There are many other bi-directional debuggers as well; see for example [172] and the references in [27]. One simple technique that is sometimes used is to save periodic checkpoints of program state, and when stepping backwards, just re-compute forwards from the previous saved checkpoint to reach the state of the program at a desired time-point.

So, alhough pure reversible computing makes bi-directional debugging trivial, it is not strictly necessary to compute reversibly in order to achieve this debugging capability. If one's only requirement is bi-directional debugging, it might be easier to just reinstrument an existing programming environment to achieve this directly, rather than coming up with a full computing reversible computing system from scratch.

10.3 Transaction processing and database rollback

It seems that the operation of "rolling back" the effects of an aborted transaction, which is common in some types of database systems, could possibly be implemented on top of a more general framework for undoing the actions of inter-communicating, reversible processes in a multitasking operating system for a reversible computer.

However, many details of this connection remain to be worked out; I can not yet say with confidence that this sort of application for reversible processing makes sense. Database rollback can already be performed quite well without requiring that the computer system be reversible at all levels. It is not yet clear whether the requirements of this application justify the sort of total, low-level reversibility that we have been discussing.

10.4 Speculative execution in multiprocessors

Similarly, it appears that reversibility might be useful to coordinate the activities of multiple CPUs which are running an underlying sequential algorithm in a parallel multiprocessing system. The individual CPUs might optimistically perform computations on data under the assumption that the data is valid (as in Knight's paper [90]), but when an inconsistency is detected, rather than restarting the processor's computation entirely, the processor might be reversibly rolled back to the point at which it read the bad data, and then proceed from there using the new, correct data.

10.5 Numerical stability in physics simulations

Apart from the performance benefits discussed in previous chapters, there are some advantages to using reversible algorithms when simulating physical systems. Reversibility is a sort of conservation law that is maintained in the real world, and so should also be maintained in the simulation. The flow of information in the physical system can and should be mirrored by the flow of information in the simulation. Failing to do this can lead to the simulated state of the system drifting farther and farther from the set of states that are possible in the real system being simulated.

We saw this behavior in our simulation of the Schrödinger wave equation (§9.5.6, p. 241). In the original irreversible version of the program, errors that crept into the wavefunction would grow in amplitude without bound. The simulation could only run for a certain amount of time before being swamped by ever-increasing artifacts in the wave function and going completely haywire. The reversible version, in contrast, although it was certainly not completely precise, always maintained a reasonably-shaped wave function, and was never observed to become swamped out by artifacts.

Perhaps this makes sense because if the artifacts steadily grow in one time-direction, then that would mean they would have to steadily decrease in the other time-direction. But such asymmetry was unlikely since the reversible algorithm was completely time-symmetric. So any artifacts that appeared could not grow unboundedly; they remained small relative to the desired wave data.

The advantages of reversibility in physical simulations are discussed further by Margolus [115]. Note however that these advantages can be gained as long as the simulation is simply reversible at the relatively high level of its state-update rule. The low-level instructions and circuits in the computer need not be individually reversible to obtain this improved simulation stability, although we saw in chapter 6 that doing so confers an efficiency advantage in large parallel systems.

10.6 Alternative applications: Conclusion

Pure reversible computing has possible applications in areas such as verifiable computation, intrusion detection and data protection, program debugging, transaction processing, and physical simulation. However, in most of the cases we have considered so far, it seems that the same benefits that could be achieved using total reversibility could be achieved using other, perhaps simpler, means as well; thus most of these alternative applications are not, in and of themselves, convincing justifications for the use of reversible computing technology.

However, if one has constructed a reversible system for other (*e.g.*, thermodynamic) reasons, then it is interesting to note that the various above capabilities fall out as a side effect. But we must remember that these alternative applications apply only if the system maintains *full* logical reversibility, but as we have seen in previous chapters, depending on the computations being performed, full reversibility may not be desirable from an asymptotic cost-efficiency standpoint. Even in our own reversible 3-D mesh model, the machine is allowed to be irreversible on its outer surface at least. Unless free energy is very expensive in a given application, it will probably be cheaper to generate some amount of permanent entropy and store it in the external universe, than it is to provide enough reversible digital storage so that a very long computation that is not inherently reversible can still be run perfectly reversibly.

Other applications for pure logical reversibility may yet be discovered, but at this time it appears that the most promising application of reversible computing technology will remain its selective use in making computation more cost-efficient by various measures; thus, that application remains the focus of our research.