One of the key issues to increasing performance in modern CPU’s is maximizing the size of the instruction window. Unfortunately, the centralized instruction windows of modern superscalar CPUs do not scale well. To overcome this limitation the WarpEngine architecture allows the large re-order buffers to be distributed. The major features of the architecture are:

- The use of tree structured control to schedule large numbers of instructions;
- Support for speculation both on data values fetched from memory and on branches;
- The ability to re-execute individual instructions when speculation fails;
- The use of explicit timestamps to order reads and writes to memory (thus allowing memory operations to occur in any order);

Other architectures, similar to the WarpEngine, include the Multiscalar and Trace processors, although they are less aggressive in the number of in-flight instructions than what we have been considering.

Any instructions that are executed speculatively may be required to be re-executed so it is necessary to save the state of all instructions. When the result of a load is re-issued (after an incorrect speculation on its memory value) the instruction at the destination of the load request is re-executed. This has a cascading effect only on the instructions on paths that are direct descendants of the re-issued load and the amount of repeated work is thus minimised.

In order to generate large numbers of instructions the control is tree structured. Thus in the code sequence in Fig. 1(a) the code for A, the conditional B and E would all be started in parallel as shown in Fig. 1 (b). This takes advantage of the fact that E will be executed no matter what the result of the test in B.

Simulation of the WarpEngine using re-order buffers of up to 10,000 instructions has taught us important lessons about how to design a CPU with very high levels of parallelism. The good news is that the parallelism is available in a range of programs including those that traverse deep and complex data structures that are not traditionally easy to parallelise. The bad news is that the amount of space needed for saving the state of in flight instructions grows more quickly than the parallelism. In many cases the resources available for storing the state are the limit on performance. The reason for this is that most code contains long chains of data dependencies that leave later instructions waiting for earlier ones to retire before they can be retired themselves. That is, the instructions sit consuming resources for a long time after they have been successfully executed but before they are retired. This phenomenon can lead to the paradoxical result that increasing the size of a problem can lead to worse performance.
Finding ways of retiring instructions out of order is one way to further improve performance. It is an open question how to do this. One approach is to take loops with independent iterations and retire them in parallel. Another is to allow user annotations to indicate blocks of code that are independent of each other (although this is counter to the philosophy of the WarpEngine which is to consider only ILP that can be extracted either by a compiler or the CPU). Another way of getting better value for state resources is to gradually retire instructions to state that is slower, cheaper, and which contains less detail about the instructions. Thus an instruction which executed a long time ago might have the details of the instruction discarded and its parameters retired to DRAM (of course there would be a high penalty for re-fetching the information should a speculation fault occur).

Using tree structured control and out of order memory access implies the need for some way to order accesses to individual memory locations. The WarpEngine does this by associating with each memory access a 32-bit timestamp. The timestamps are in the same order as a sequential execution of the instructions. Generating such finite length timestamps has turned out to be challenging. A naïve way to do this would be to make each timestamp a binary string recording the pathway from the root of the control tree to the instruction. This runs out of precision as soon as the tree is 32 levels deep. One solution we have closely examined is the inclusion of auxiliary instructions that estimate the total size of the control tree below each node.

The results of these investigations are clear. Firstly, it is essential to attempt some form of estimation or else the timestamps quickly become exhausted. On the other hand it is impossible to get it exactly right. That is, there will often be many active branches in the tree where the total amount of work to be done is potentially unbounded. One benefit of estimating the amount of work to be done is that it makes allocating resources for new instructions much easier. The problem here is that as each branch in the control is expanded it needs to be inserted somewhere in the re-order buffer. If there are already instructions in flight before and after the new code then it is not easy to work out where to slot it in. However estimates allow space to pre-allocated for the code to slot into. Incidentally this provides a strong contrast between the WarpEngine and the Trace and Multiscalar processors that are unable to insert new instructions in between ones that are already in flight.

Simulations for parallel AVL tree insertion on the WarpEngine containing a reorder buffer of 16,000 instructions is shown in Figure 2. The graph shows that an IPC of 30 has been achieved. It is worth noting that AVL tree insertion is difficult to parallelize using explicit synchronisation techniques as any node in the tree can be modified by a rotation, so it is necessary to lock all nodes in the tree from the root to the leaf where insertion is done. As the WarpEngine does not require locks, allow it is able to extract the maximum amount of parallelism.

Our conclusion is that it is possible to build re-order buffers with thousands of instructions and to use them to extract significant amounts of instruction level parallelism. But to do this efficiently and effectively requires careful attention to the issues of compact storage of saved state and to the estimation of resource usage down different control paths. Further details on the WarpEngine project can be found at http://www.cs.waikato.ac.nz/timewarp/wengine/.