Summary Part 2
Compiler Optimization of Scalar Value Communication Between Speculative Threads

Mrinal Nath (ID: 3307043)
February 9, 2005

Did this paper address an important issue? Explain.

In TLS significant amount of time is spent in failed speculation i.e. epochs have to be squashed and restarted due to data dependence violations. It is the hardware which tracks these dependences and detects violations. If the inter-epoch dependences are quite frequent, then these failed speculations can become a performance bottleneck. In such cases, it is better to stall the dependent epoch till the dependence is resolved. However, the hardware usually does not have enough "foresight" due to the limited instruction window. This is where the compiler can help, since its instruction window is virtually infinite. The compiler can explicitly synchronize reads with writes by using wait and signal instructions.

However, if the insertion of these synchronization instructions is done in a very simplistic manner, successor epochs may waste a lot time being stalled for data values to be produced by predecessor epochs. This is because the forwarding path is long. (Since these paths limit performance, they are called critical forwarding paths.) This is the problem that this paper is attempting to solve. This paper focuses particularly on optimizing the synchronization of scalar (register-resident) value between epochs. This is an important problem, since there is potential for significant performance improvement if the time spent in synchronization can be minimized.

Are the proposed approaches valid? Describe its strength and weakness.

Description of approaches and strengths:

The techniques proposed in the paper are targeted only at loops. Loops in a program are selected for parallelization by the compiler only if certain criteria are met. These criteria are chosen such that significant parallelism is likely to be achieved by parallelizing these loops. The compiler inserts the synchronization instructions based on some simple constraints, which are required to ensure correctness of the resulting code. A placement algorithm is developed which tries to place the wait statements as late as possible and the signal instructions as early as possible in an epoch. This algorithm is based on a dataflow analysis of the CFG of the epoch.

Even after applying the basic algorithm, there is much scope for further improvement. To optimize the scalar value communications further, two algorithms, based on dataflow analysis, are proposed. One is a ‘conservative’ algorithm, which schedules instructions such that the signal instruction and the other instructions that it is data dependent upon, are moved as early in the epoch as possible, without scheduling across branches or other data dependences. This technique can perform better than hardware based synchronization approaches, and thus can reduce the hardware complexity.

The other algorithm is an ‘aggressive’ version of the first algorithm. This algorithm is capable of speculatively moving the signal and associated instructions across branched and/or data dependences. The neat idea here is that some paths are ‘frequent’ while others aren’t, so it makes sense to speed up
the frequent path while accepting some penalty if the execution follows the infrequent path. To this end, the compiler (using information gathered by profiling) moves the signal and associated instructions ‘upwards’ along the frequent path. Some recovery instructions are inserted in the infrequent path, thus incurring a penalty on this path.

**Weaknesses**

The techniques depend heavily on profile information to make good selections of loops for parallelization. If the actual input differs very significantly from the input set used for profiling, then the performance may not be as good, or may even be poorer. The aggressive scheduling algorithm does not always perform very well, and it also required hardware support for successful recovery, if the execution follows the infrequent path. Also, best performance is achieved if the aggressive algorithm schedule past both data and control dependences. Also, the reduction of the synchronization period exposes new dependences and increases the number of failed speculations.

**Do the results support the conclusions? Explain.**

Yes. The results show that the compiler is very successful in scheduling code so that the total time spent in synchronization is reduced. This leads to good speed up in the parallelized parts of the code. Further, since these parts were carefully selected to be a significant proportion of the program runtime, the overall program runtime is also improved. The comparisons against hardware approaches, and in conjunction with hardware approaches shows that the conservative scheduling algorithm can remove the complexity from the hardware, since the compiler performs much better than the hardware. The results also show that the aggressive scheduling algorithm should be used selectively, since it not only requires hardware support, but may hurt performance in some cases.

**Describe the potential future works?**

This work was targeted towards loops only. This work can be generalized to other regions of code. This paper deals with scalar value communication between epoch and the other paper deals with memory-resident value communication. There should be a study of the advantages and limitations of applying the two optimizations (for scalar as well as memory-resident) simultaneously. Intuitively, this should give good performance, but there may be other side effects (like exposing previously hidden dependences) which will limit the performance improvement. Also, since we know that after applying this optimization, new dependences are exposed, there may be some scope to apply multiple “passes” of this optimization to further reduce the effects of the newly exposed dependences.