Summary Part 1

Compiler Optimization of Scalar Value Communication Between Speculative Threads

Mrinal Nath (ID: 3307043)

February 7, 2005

What are the problems solved by this paper?

Traditionally, multithreaded processors are used to improve the throughput by running independent threads or processes simultaneously. But if such a processor is given a single ‘normal’ program, then it can’t exploit any TLP, since there isn’t much explicit TLP in a ‘normal’ program. Such programs are also very difficult to parallelize (especially integer programs).

TLS allows the compiler to speculatively create parallel ‘epochs’. The hardware must support TLS by ensuring that data dependences are not violated. If an epoch violates data dependences, then it is squashed and restarted. This can be a significant overhead, especially if the inter-epoch dependences are ‘frequent’. In such a case, it is better to stall the dependent epoch than to speculatively execute it. But if the dependent epoch is stalled for a large part of its execution time (near serial execution of epochs), then there isn’t much performance gain from using TLS.

If the critical forwarding paths in an epoch can be shortened, then more parallelism can be achieved between the epochs, giving good performance improvement. This is the basic issue being tackled in this paper.

What are the approaches attempted by this paper?

Explicit synchronization between scalar value uses in different epochs can be achieved by using wait and signal instructions. The simplest idea is to place waits at the beginning of the epoch and signals at the end of the epoch. But this will lead to the serialization mentioned above. This paper describes a basic optimization, by which the waits are placed as late as possible (before the first use of the scalar) and the signals are placed as early as possible (after the last definition of the scalar) in the epoch.

It is found that after this optimization, there is still scope for reducing the critical forwarding paths. A ‘conservative’ algorithm is described which moves the signals and the related code as far up in the epoch as possible (the related code may have to be duplicated or may get combined). This algorithm does not move the code beyond dependences. An ‘aggressive’ algorithm is also described. This algorithm can move code beyond dependences, but requires additional hardware support for validation and recovery.

What are the main conclusions of this paper?

It is found that these compiler techniques give good amounts of program speed up, since the critical forwarding paths are shortened and more parallelism is exploited.

The ‘conservative’ algorithm gives good performance improvement for many of the benchmarks. An important finding is that hardware mechanisms for shortening the critical forwarding paths are not necessary if this compiler optimization is used. The ‘aggressive’ algorithm works well only in some cases and hence it should be applied selectively. Otherwise there may be performance degradation due
to the associated overheads. Also, it works better if the code movement is across both control and data dependences.