Summary T5

This week we discuss a paper dealing with "Speculative Decoupled Software Pipelining". It bases on last week's paper, but due to the fact that DSWP is not mighty enough to parallelize many loops it expands this approach by using speculation.
The problem that arises when certain loops are parallelized is that there are dependence recurrences inhibiting a parallelization using DSWP. To solve this problem thread-level speculation could be used to break special dependences speculatively, but this would also enlarge the latency on the critical path because of the necessary communication. SpecDSWP can avoid this enlargement by keeping the organization pipelined.
In cases of misspeculation the approach provides a mechanism to recover the program state. For this purpose threads create a kind of snapshot before executing a new iteration. When misspeculation is detected every thread restores its changes to memory and the registers and continues executing the iteration sequentially.
In order to find the edges in the dependence graph the algorithm speculates all highly predictable dependences provisionally. Then threads are allocated to the remaining edges in the PDG using a partitioning heuristic. After that it identifies all provisionally speculated edges that crosses the thread border and at the same time points from a later thread to an earlier one in the pipeline. The edges that are found this way are exactly the edges that need to be speculated.
A likely use-case for SpecDSWP are biased branches. This means that one branch of a condition is taken in most cases and the other branch only rarely. In such a case the compiler can break the control dependence. Technically this speculation is implemented as an unconditional branch replacing the conditional one. To detect misspeculation a boolean flag is set when the unlikely branch is taken and the rollback-mechanism re-executes this branch sequentially.

Questions / Opinion:

1. I wonder how to implement a misspeculation detection or a value predictor in hardware.

2. How can I imagine those heuristics they mentioned in relation to the partitioning?

1 Summary

In this paper, the authors present a technique combining Decoupled Software Pipelining with speculation. Speculative DSWP focuses on breaking dependence recurrences by speculating dependence recurrences in loops, to allow performing speculative DSWP on loops, where normal DSWP would not be possible.
DSWP partitions the loop body and distributes parts to cores. A core is then responsible for its partition in all iterations of the loop. DSWP constructs a PDG and detects dependence recurrences by identifying strongly-connected components. By speculatively ignoring specific dependences, the strongly-connected components may be split into smaller ones.
SpecDSWP uses hardware versioned memory to rollback on misspeculation. Misspeculation detection is realized in software. SpecDSWP features speculating dependences, as long as an appropriate misspeculation detection can be employed, and value prediction.
SpecDSWP furthermore speculates the execution of infrequently executed basic blocks, which can break even more dependences.
The SpecDSWP compiler identifies frequent silent stores, and guards them by first checking if the written value equals the current value. The store is only performed, if the values differ. This allows further branch speculation.
The SpecDSWP benchmark uses a synchronization array as a low-latency queue and a versioned memory system supporting 32 outstanding versions.

2 Questions

In the experimental results the authors compare their approach to single threaded code. It would be important to know whether this code has already been optimized by DSWP. Especially for the `gzip` benchmark one cannot say whether their approach caused a speedup compared to the normal DSWP variant.

Speculative Decoupled Software Pipelining

1 Summary

This is an optimization on the paper discussed last week. Authors add speculation on DSWP. DSWP alone is not useful to parallelize programs with complex loop structures. So, speculation is used to break loop carried dependence, and extract additional parallel segments.

Speculation is made by assuming that some apparent dependences are virtual. So compiler first builds the PDG for the loop. Then identifies which dependence edges from PDG can be removed. Few heuristics are discussed to carry out this. Biased branches in the loop body are speculated. These branches are replaced by unconditional branches with a misspeculation detection scheme. Similar heuristic is used for speculated infrequently executed basic blocks. Another speculation is silent stores. Some kind of analysis is performed on the program to detect the stores that never happen or happen infrequently. All the output dependence and anti-dependence edges are also removed from PDG.

Misspeculation recovery is done using the snapshot of the system stored before starting the execution of particular iteration. Whenever a misspeculation is detected, that particular thread waits for completion of all previous threads. Then stored snapshot is recovered before starting the execution of non-speculated version of misspeculated thread. After this step, speculative execution can be started again. Versioned memory used in this scheme is similar to transactional memory in that it can be committed in case of success or rolled back safely in case of misspeculation. However, there are some differences also like absence of conflict detection in versioned memory

Misspeculated iteration can be re-executed either on a single thread or multiple threads. But communication overhead involved might cancel out the gain in case of multiple threads.

2 Questions

1. In addition to synchronization array, there is also need of storing snapshot for each iteration. Isn't this approach less memory efficient?

2. I do not understand the misspeculation detection scheme used for biased branches.

Summary Speculative Decoupled Software Pipelinging

The paper extends the previously discussed DSWP with speculation to increase both its applicability and its effectiveness.

The authors start by comparing DOACROSS and DSWP parallelism; stating that DSWP can perorm better as it tolerates variable latency in threads. They then show an example that can be neither parallelized by DOACROSS nor by DSWP, but that can be once one adds speculation. They argue that, as before, DSWP can perform better, as dependece communication is not required on the critical path (as opposed to DOACROSS). The actual DSWP part is quite similiar to the one we saw in the last paper, including the synchronization array. There are however some differences/additions: The start by computing strongly connected components of the PDG; thos build their basic scheduling units, and they are distributed among a number of threads.

Speculation is used to shrinken the SCCs, in the hope of gaining better scalability/performance. The system speculates to remove edges from the PDG; in a first step, it only speculates on those that are "highly predictable"; it then identifies their subset which is cross-thread and only speculates on those (removes them). Besides such control flow speculation, they also speculate on values: They first profile the application to find frequent silent stores; and then speculate on them keeping their value. As speculation can fail, the system can detect such failures and to recover from them. Missspeculation is detected by adding abort code to the branch which was predicted not to be taken. For its handling, the authors introduce a commit thread. For each iteration, it waits for all threads signaling it their status (exit, misspec, ok). If all threads sent ok for an iteration, the corresponding memory state is commited to "real" memory. If not, recovery is initiated. During recovery, after shutting down further speculative execution, the memory state is reset to the last sane state, and all inter-thread messages are discarded. Then, the code which caused the failure is reexecuted non-speculatively - either sequentially by the commit thread, or by the worker threads, but with synchronization between them. The memory rollback is a cheap operation, as the authors added a so-called version memory, a hardware extension, which augments loads and stores with a version number.

The authors evaluate their idea on simulated harware. In this case, it smells fishier than usual, as they state that "the detail [. . .] prevented whole program simulation".

Open Questions:

- Why is special handling required for callee-saved registers?
- How exactly does the polling of the statuses in the commit work and what is its overhead?

Summary T5

This week's paper is an adaptation of an approach the last paper was based on. The parallelization approach is called "Decoupled Software Pipelining" (DSWP) but as it works non-speculatively it's capabilities to parallelize everyday programs is very limited. The authors of this paper extend DSWP to work *speculatively* and introduce SpecDSWP.

The basic concept of DSWP is not changed. Instead, the approach tackles the dependence edges. Obvious dependences are speculated and a heuristic tries to remove further dependence edges. This ensures that DSWP is able to parallelize code it was not able to parallelize before. But, to ensure correctness in case of misspeculation, the code must be able to rollback.

Concerning loops, SpecDSWP saves each completed iteration as a state the system rolls back to in case a misspeculation was flagged. The rollback mechanism works the following way: the system must wait until the thread(s) executing the previous iteration finish. It then restores the old memory values and a non-speculative version of the iteration is executed. After that, speculation can continue.

SpecDSWP also speculates about how often branches will be taken. In some cases this is able to resolve further dependence-edges, which in turn leads to more possible parallelizations.

An important fact I forgot to mention is that SpecDSWP works on versioned memory. Versioned memory is a technique to store multiple versions of values at the same address. This means, that values are accessed with version-address tuples instead via only the address. Each thread manages a version number which is increased in each iteration. In case of a rollback, everything from the version number being rolled back to all the other dependent versions is discarded.

Open questions:

1.) In Figure 7, Thread 4 (7(b)) also consumes nodes. Why does it need to do that when Thread 3 (7(c)) already does that? Wouldn't that in fact yield erroneous code since Thread 4 might consume the nodes which were meant for Thread 3?