Concept Chapter3 Chapter 3. Concept

From a high point of view SPolly is divided into a speculative loop parallelizer and a non speculative extension to Polly. Even if the objectives for both parts are the same, namely to improve the performance of loops, the approaches to accomplish them are different. While the former one will introduce speculative parallelism for promising loops at runtime, the later one tries to weaken the harsh requirements on SCoPs in order to make Pollys loop optimizations applicable on a wider range of loop nests. In the presented setting both approaches may benefit from the polyhedral optimizations and also from parallel execution, so it is hardly surprising that the polyhedral analyses play a decisive role. On the one hand they reveal loop nests which may be optimized by Polly, with or without the extensions of SPolly, on the other hand they are used to detect promising loops to speculate on. Apart from the implementation work, which will be described in the next chapter, this thesis presents the concepts and key ideas behind. We believe that these ideas and the gained knowledge is very valuable not only for future work on SPolly or one of its bases but also for other approaches facing similar situations.

Speculative Parallel Execution SpeculativeParallelExecution Speculatively executing loops in parallel is one of the two major purposes of SPolly. The challenge was to exploit as much parallelism as possible without restricting the applicability to much.

s Even if the execution is secured by a (not extended) STM it may not be sound to execute a loop nest in parallel. If there is a conflict between two iterations and the later one (in terms of the original ordering) commits its result before the former one could, the STM will force the former one to recompute, but now based on corrupted data already committed and therefore permanently written. From this fact the need of an commit order arose, as it would in any case preserve the initial ordering when applying permanent changes.

As one great feature of Polly and the underlying polyhedral model is to detect and model loop carried data dependencies, it sounds plausible to use this for speculative execution purposes as well. Unfortunately speculative extensions will compromise this ability because they will speculatively assume less dependencies in order to weaken the requirements. To handle such situations SPolly tries to unify both of its parts. At first, non computable memory accesses, as they arise from aliases we cannot check in beforehand or from function calls within a SCoP, are overestimated. The resulting polyhedral model is complete with regards to possible data dependencies, but could anyway reveal transformations for a better data-locality or possible vectorization. Afterwards speculation is applied, thus the optimized loop nest is speculatively parallelized. Note here, that only Polly could change the iteration order but because this would be based on a complete and overestimated polyhedral model, it would still yield a sound ordering.

In the worst case there are dependencies between all parallel executed iterations and each thread has to recompute its part once another one committed its results. As countermeasure to such time consuming misspeculations SPolly monitors the runtime of all speculatively parallelized SCoPs and reverts the speculation for poorly performing ones.

In contrast to the speculation free optimization approach, this one is also capable of parallelizing sSCoPs with almost arbitrary function calls. To do so the STM has to provide the wait construct as described earlier. Even if this is not yet the case, SPolly could introducing such wait with no further effort. This will certainly cause additional overhead during the parallel execution, but it does not preclude a performance gain per se.

In the special case of sSCoPs, we know certain things about the control flow and the memory accesses within a parallelized region. With the assumptions we may derive from the requirements on sSCoPs it is possible to secure the speculative parallel execution by introducing wait constructs right before each possibly violating function call. Executing those function calls in the correct ordering will enforce all their possible effects to be applied in the correct order too. If arbitrary control flow and conditionals are allowed, this conclusion does not hold anymore. In fact every memory access could change the taken execution path of another thread, thus every branch would need annotation.

Speculation Free Optimizations SpeculationFreesSCoPs As second objective SPolly may enable sound polyhedral optimizations, even parallelization, on sSCoPs with no need for speculation at all. This is the case when the tests we introduce in the following can rule out all violations of a sSCoP.

SPolly In A Nutshell

wrapfigure[]r0.4 *-2mm [width=0.4]Figures/draftPaperCT.eps Draft paper:

*-5mm fig:draftPaperCT