Index Set Splitting

Index Set Splitting is a preprocessing phase for algorithms in the polytope model which yields schedules which are, in some cases, orders of magnitude faster. These are cases in which the dependence graph has small irregularities. The idea of this algorithm is to split the domain of loop nests into parts and apply existing algorithms to produce schedules to there individually. For the case of uniform dependencies it is already proven that there exists algorithms which yield a schedule with (asymptotically) optimal latency. But for afiine, non-uniform dependence the optimum is sometimes missed by orders of magnitude. The use of different schedules for different iteration domains improves this situation.

The first important definition is the latency of a program. It is the difference of the maximal and minimal value of the scheduling function. This can be interpreted as running time on a parallel computer with enough processors. It is clear that we have to minimize the latency in order to get a possible better execution. This concept can also be extended to individual statements. It's a lower bound on the latency of the whole program.

A first naive approach to split the index set is to subdivide the iteration space for a statement S where each part corresponds to a selection of the incoming dependencies. We need to consider all paths from a statement R to this statement S. This means we have to consider d2 o d1 if there is a dependence d1 from R to T and d2 from T to S. If the graph is a tree we have no problem but if the graph is a directed cyclic graph there can be O(2^l) paths which then have to be merged. This would yield in a doubly exponential algorithm and is therefore impractical in practice.

The solution is to associate a finite automaton with the dependence graph and use well known algorithms for associating any two states S and T with a regular expression, representing all paths from S to T. With this first version we lose all information on the delay associated with the composite dependence and can't produce good results. To circumvent this problem we consider additionally every single dependence once and split the index set accordingly. This holds some information which is lost during the transformation.

The described algorithm treats the program as a whole and searches for all potentially useful splits. For large programs it is sometimes desirable to apply index set splitting only when and where it is necessary. The basic idea to fix this is to first schedule the program with standard techniques and if we want to improve the program use index splitting then. We could also change the algorithm to take resource constraints into account. If we are already using all processors it is useless to increase parallelism.

Open questions:
1) Is this used in practice?
2) How to generate code for this different index sets?
3) Hot does the automaton stuff work?

Summary of "Index Set Splitting"

1 Summary

Space-time mapping methods for the automatic parallelization of loop nests related every instance of every statement to virtual point in schedule and allocation. We have some problems of schedule computation for affine, non-uniform dependences due to magnitude order. The main idea of this paper is to partition the index sets of all statements into fixed number of parts, and compute the schedules for each part.

Given a dependence graph, nodes are statements, edges are dependences, and each node S is given a subset $I(S)$ of $\mathbb{N}^{p_s}$, here p_s is the nesting depth of S. $I(S)$'s elements are called iteration vectors. We will do our algorithm processing on this graph.

A naive splitting algorithm simply compute all possible paths in the dependence graph, and split every statement according to all incoming paths. The problem of this is the complexity of merging the splits of k incoming dependences cause exponential 2^k complexity. The number of paths cause algorithm exponential then impractical.

Effective algorithm consider to abstract from precise paths in order to avoid the exponential happening. Replace the dependence graph with automaton, whose states are statements and transition are dependences.we can compute composite dependence by Omega calculator if we use regular expression to representing all the paths from state S to state T. The heuristic idea is to consider every single dependence once, and split the index set of target statement into range and the rest, which guarantees the properties of different dependences are considered at least once but complexity only increased by linear factor instead of exponential number of paths. Intuitively, if we split the target index set $I(S)$ of every single dependence d into the range of d and the rest then we have a approximation of the range of all paths to S whose last dependence is d. Then idea is to propagate these ranges of d along every path to all other statements.

Summary of splitting algorithm as following:

1. For all dependence d to compute a as small as possible polytope R_d containing the range of d then split the index set $I(T)$ into R_d and $I(T) \setminus R_d$.

2. Using Kleene's algorithm to give description for all paths

3. For every pair of Statement (T, S) and every dependence d with statement T and every path p from T to S do: interpret the path description p to

map points of index space $I(T)$ to points of $I(S)$, and compute image $p(R_d)$. This will divide the index set $I(S)$ into a part which is in the image of R_d under p.

4. For any statement S, combine all splits obtained in this way

For large programs, we don't need to search all potentiality useful splits but only when and where is necessary. the idea is trying to schedule the program without splitting, if result is not satisfactory, then use index set splitting to improve the solution.

Generally we can adapt our method as a trading off quality of parallelism for analysis time.

- Search for more splits but the search cost doubly exponential.
- We can turn off the propagation phase in order to save compilation time
- Improve the solution by propagating a split through cycles more than once.
- Only split when these statements are responsible for bad quality of schedule.

2 Questions

- Kleene algorithm to give the regular expression to present paths unclear?
- Only do the potential useful split, how do you know it is necessary?

Summary for

Index Set Splitting

The paper by M. Griebl, P. Feautrier and C. Lengauer introduces index set splitting as a pre-processing state for polytope model optimization. For regular/uniform dependences, it is possible to determine an asymptotically optimal schedule. For non-uniform dependencies, however, an optimal schedule cannot be constructed in general. The idea of this paper is to partition the iteration domain into regular subsets, which can then be scheduled independently of each other.

The authors first introduce a simple, naïve implementation of an index splitting algorithm. Intuitively, the algorithm tries to divide the iteration space for one statement by the incoming dependencies (i.e. the flow to the statement). An algorithmic approach to this intuition would compute all possible paths to any statement, then splitting the index set of every statement according to those paths. A more thorough analysis of this algorithm suggests a few optimizations: There are always multiple paths to any statement, which might or might not align at some point. Therefore the split must encapsulate all paths which share a common suffix. This problem is expensive, since enumerating any paths and merging the splits takes runtime exponential in the number of incoming dependencies. Even worse, the number of different paths connecting two nodes is exponential, even in the absence of cycles. Hence, the algorithm is most likely too inefficient in practice.

In order to find a finite (and more importantly, input size independent) description of such paths, the authors propose to depart from the polyhedral description in favor of FSAs. In particular, any set of paths between two statements will be associated with a regular expression. Still, this description is not obtained without a loss of precision, since the transitive closure of affine relations is not necessarily affine itself. The authors proceed to deploy heuristics to deal with such cases. The proposed algorithm will keep all dependencies in mind, and split the index set into the set of its targets and the rest before proceeding further along the description. This gives a conservative approximation on the minimal range of each dependency, which is propagated along.

The authors introduce yet another optimization heuristic. While the proposed method is powerful, it is not equipped to deal with problems on a very big scale, e.g. large programs with only very few instances where index set splitting would be useful, or small scale optimizations, were the polyhedral description with its higher precision would most likely yield better result. Hence we can use simpler, but faster, scheduling algorithms first and then apply index set splitting for dependencies which could not be resolved. Similarly, if the loss of precision incurred by index set splitting gets too large for smaller problems, transitioning to the polytope model for the partitioned index sets will most likely yield a better solution.

The paper presents the basic idea to partiston the index sets of all the statements independently of the problem size into the fixed number of parts and compute individual schedules for each part.Since it is not genreally not possible to find the arbitary schedule for minimum latency, so one should restricts the search to a subset of all possible functions , a function is affine in loop counters with unknown coefficients. which then can be solved with the linear programming methods, but for some problems , resulting linear program can be infeasible then one can resorts to multidimensitional schedules.

Index set is diffrent from that tiling that inscreases the granularity but index set spliting does not change number of dimensions but benefits from an individual treatment of the various partitions.

Paper also gives introduction about the naive method of spliting , which found to be ineffective if there are loops in the statement dependence graph since, within the strongly connected components, the number of paths are unbounded.

Spiting algo says that

1. For the all dependences d, compute a polytope R_d containing the rande of d, and the split the set I(T) of the target statement of d into R_d and $I(T)/R_d$.

2. Compute a description for the set of all the paths in the statement dependence graph in kleene´s algo.

3. Then find realtion path that maps point from index space I(t) to I(S) and compuet image $p(R_d)$ of this relation. Then we divide the index set I(S) into a part which is in the image under p and the rest.

But few considerations in practice: for large dependence it is worthless for uniform dependencies and due to the condensed description of set of all paths and overestimation of the reflexive transitive closure by omega we lose precision and does not find the split.

For large programs it is generally advised to apply split only when needed , .i.e. first schedule the program without spliting and if result is not satisfactory , then improve by spliting. however this algo also fails when statements which have more parallel instances than there are real processor avaliable. here we are not interseted in optimal solution but by concentrating on the order of the magnitude by which running time is reduced.

One must also consider the tradeoff between the number of split and cost of expotential search. and we can skip propogation step to save the compilation time at the cost of some useful splits.

Questions:

1. What are uniform and non-uniform dependencies?

2. Since we can´t decide whether the spit is useful locally, we accept useless split(by uniform dependences)how we ensure that spiting by non-uniform dependencies are always good.

3.Cubic description of the paths(by kleene,s algo) is loss of precision why not try some other way like Dijkstra's or Johnson's algorithm.

4."Turn-off propogation to save compilation time at cost of some useful split", and what is more important: saving compilation time which happens once or parallization of code by useful spliting ,which runs several times?

Index Set Splitting

Index Set Splitting is a technique to minimize the latency of schedules in the polytope model, or in other words, to maximize the parallelism of a loop nest. In order to do this, the index sets of statements within the loop nest is literally splitted (as in loop splitting). This splitting is done to remove (irregular) dependencies, thus the resulting polytope (with less dependencies) might allow more parallelism.

Multiple algorithms are presented but they are all based on the same idea. First compute the dependencies between statements e.g., by computing a dependency graph as other scheduling algorithms do. For a dependency (or a composite of dependencies) which ends in a statement T, split the index space of T into the image of a such a (composite) dependency and the remaining part. After splitting there will be an instance of the statement T in the image of the dependency which has to satisfy the dependency and one in the remaining part of the iteration space, for which the dependency is effectively removed. After the splitting step is done e.g., all dependencies are split as described or a heuristic ends the process, the resulting polytope is (re)scheduled.

Splitting all (composite) dependencies will result in a exponential algorithm with regards to the number of dependencies between to statements, in the words of the authors an unpractical approach. To counter this growth they propose to use regular expressions to express (and effectively merge) dependencies in the graph. Detailed information (the exact path) is lost while the number of paths is reduced to one per statement pair. As they state this approaches lacks to much precision they combine it with the initial one. All (single) dependencies are considered once before the combined ones are considered in the simplified way. As the image of any composite dependency is part of the image of the last dependency on the path this will conservatively approximate the image of each composite dependency considered by the initial exponential algorithm.

For their final algorithm version they first use a non-splitting scheduler and analyse the result. In order increase parallelism, statements are only rescheduled if splitting yields a smaller latency and the not all processors are already assigned a parallel instance of this statement yet. This version (with some technical modifications) was implemented in the LooPo parallelizer but without giving actual results on performance or runtime.

Questions

- Why are there two schedules given for the first example, or two be more precise whats a latency schedule and how is it computed?
- A multidimensional shedule could increase locality for example 1 and allows parallel execution as well. Simple splitting will transfer the challenge to find this to the sheduler afterwards. As this is a difficult chalenge, could the information (gained during the split process) be used to help the scheduler?
- Whats $d_{upwards}|d_{downwards}$ and d^+ ?