Does dynamic and speculative parallelization enable advanced parallelizing and optimizing code transformations?

Philippe Clauss, Alexandra Jimborean

HAL Id: hal-00664339

https://inria.hal.science/hal-00664339

Submitted on 30 Jan 2012

Does Dynamic and Speculative Parallelization Enable Advanced Parallelizing and Optimizing Code Transformations?

Philippe Clauss and Alexandra Jimborean, CAMUS group, INRIA, LSIIT, University of Strasbourg

Thread-Level Speculation (TLS) is a dynamic and automatic parallelization strategy allowing to handle codes that cannot be parallelized at compile-time, because of insufficient information that can be extracted from the source code. However, the proposed TLS systems are strongly limited in the kind of parallelization they can apply on the original sequential code. Consequently, they often yield poor performance. In this paper, we give the main reasons of their limits and show that it is possible in some cases for a TLS system to handle more advanced parallelizing transformations. In particular, it is shown that codes characterized by phases where the memory behavior can be modeled by linear functions, can take advantage of a dynamic use of the polytope model.

1. INTRODUCTION

The advent of multicore processors puts a high pressure onto the software: it must exhibit sufficient parallelism in order to allow the available hardware resources to be converted into significant performance gains. The lack of an efficient compiler technology becomes evident, with the urge of executing general-purpose software on multicore platforms. Particularly, it is a challenging task to parallelize code at runtime, if the information available at compile time is not sufficient.

A well-researched direction for overcoming these difficulties and parallelizing general-purpose applications is thread-level speculation (TLS) [Patel and Rauchwerger 1999; Cintra and Llanos 2003; Chen et al. 2003; Quiñones et al. 2005; Johnson et al. 2007; Raman et al. 2008; Oancea et al. 2009; Raman et al. 2010; Tian et al. 2010b]. A TLS framework allows optimistic execution of parallel code regions before all dependences between instructions are known. Hardware or software mechanisms track register and memory accesses to determine if any dependence violation occurs. In such cases, register and memory state are rolled back to a previous correct state and sequential re-execution is initiated.

Unfortunately, most TLS proposals have yielded only modest performance gains, or have often been based on hypothetical hardware mechanisms and simulators [Quiñones et al. 2005; Liu et al. 2006; Johnson et al. 2007; Raman et al. 2008; Oancea et al. 2009]. One major limitation of such work is that parallelization is attempted on unmodified code generated by the compiler: when considering loop nests, the strategy usually applied is to cut the outermost loop into contiguous chunks and run these chunks separately in multiple threads. Unfortunately, as soon as a dependence is carried by the outermost loop, this approach leads to numerous rollbacks and performance drops. Moreover, even if infrequent dependences occur, nothing ensures that the resulting instruction schedule leads to significant performance gains. Indeed, poor data locality and a high amount of shared data between threads can yield a parallel execution slower than the original sequential one. Moreover, it is well-known that the
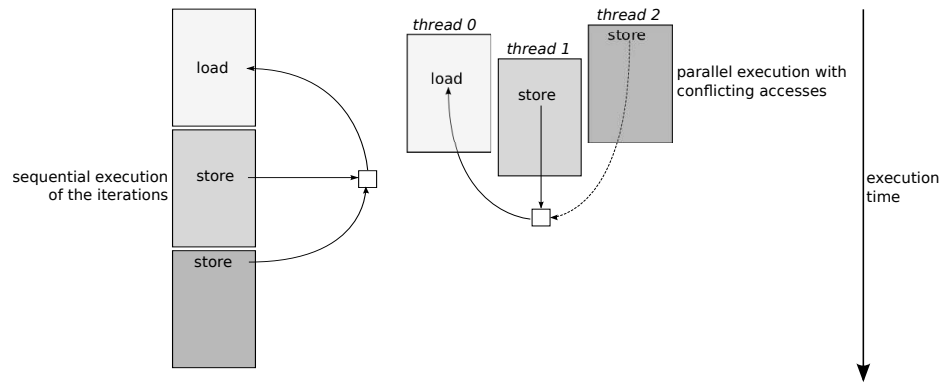
Fig. 1.   Illustration of the usual TLS parallelization by chunks and conflict detection

performance of parallel code is very sensitive to many parameters, and fine-tuning is often mandatory.

On the other hand, Prabhu and Olukotun looked at manually exposing thread-level parallelism (TLP) in Spec2000 applications [Prabhu and Olukotun 2005]. They showed that substantial increases in TLP were possible using a variety of transformations for traditional sequential applications.

All these facts argue that even if TLS seems to be a relevant approach to achieve automatic parallelization of general-purpose codes, it remains inefficient as long as optimizing code transformations cannot be handled at the same time. This paper discusses in Section 2 the reasons why current TLS systems have such limited features, and proposes in Section 3 a way to overcome these limitations when the targeted code exhibits a linear memory behavior. The proposed approach is evaluated on four synthetic benchmarks in Section 4. Conclusions are given in Section 6.

2. TLS SYSTEMS LIMITS

2.1. Why limited parallelizing transformations in TLS systems?

Any speculative system requires to verify all along the execution of the parallelized code that it stays semantically correct, in order to validate or invalidate its execution. In TLS systems, a given thread can be responsible of a suspicious memory or register access that has to be canceled by performing a rollback to a previous safe state.

When the parallelization strategy consists of cutting a loop in contiguous parallel chunks, each chunk being run by a different thread, verification is achieved by directly comparing the memory behaviors of the parallel and sequential versions, the latter, obviously, being the baseline. It consists of monitoring the memory and the register accesses of the speculative threads, in order to verify if accesses made by different threads to the same memory locations occur in an order which is different than the original sequential order. Since the i-th thread runs code that would be run before the code of the $(i + n)$-th thread in the sequential version, any write access from the $(i + n)$-th thread to a common memory location and occurring before any access from the i-th thread generates a rollback of the $(i+n)$-th thread in the parallel version. This mechanism is illustrated in figure 1.

This verification scheme is relatively obvious, since the parallel execution can be directly mapped on the original sequential order.

Notice that significant improvements have been provided in some proposals to this parallelizing strategy, in order to apply to a larger set of codes. For instance, value prediction is used when a dependency occurs between threads, in order for each thread

to have a better chance to be validated at the end of their run [Raman et al. 2008; Tian et al. 2010a]. However, even with such improvements, it is always the same parallelization strategy which is applied: contiguous chunks executed in parallel.

This is mostly due to the fact that if the original code is transformed by significantly re-scheduling its instructions, it becomes extremely difficult to identify any conflicting memory or register accesses, since the parallel execution order can no longer be easily mapped on the original sequential order.

2.2. Understanding more formally the speculative parallelization issues

Program analysis theory for code parallelization provides the well-known notion of dependences between instructions. In parallel programming or compile-time automatic parallelization, any code transformation is semantically correct if all the dependences are respected: two dependent instructions that have to be executed successively are also executed successively in the transformed code.

In speculative parallelization, dependences are unknown before the code has been run. However, any parallelization is considered as "valid", since any dependence violation is recovered thanks to the rollback and the transactional memory system. But too many dependence violations induce a huge overhead from the numerous rollbacks. Hence it is still essential, if possible, to get dependence information as soon as possible to generate efficient parallel code.

A previously proposed approach is to run in advance a subset of the code which performs just the memory address computations, in order to get the information regarding all dependences. This idea is used in the inspector/executor model, where sequential programs are divided in two components. The first one, called the inspector, is in charge of extracting the program dependences, usually between loop iterations. Then, an executor runs the tasks as soon as all their dependences have been satisfied. This model has been first proposed by Zhu and Yew in [Zhu and Yew 1987], and has been later extended in many directions [Chen et al. 1994; Rauchwerger and Padua 1995; Michael Philippsen et al. 2011].

In general, this model is efficient if the address computation is clearly separated from the actual computation, allowing the creation of an efficient inspector. Moreover, to capture the dependences with no restriction, some control bits are commonly associated to every array element during the inspector phase. This often restricts those methods to array accesses, and can lead to major memory overheads. Moreover, pointer references can strongly disturb the automatic inspector creation, limiting the applicability of this method.

Hence it is most often impossible to validate a code transformation in advance. Following the general idea of verification in speculative systems, the transformation should be verified while the transformed code is being run.

2.3. How to validate a speculatively parallel code while it is running

Code transformations are usually guided by sufficient knowledge of the code properties and by objectives like data locality optimization, energy saving or load balancing. In the speculative approach, transformations should also be guided by predictions on initially unknown code properties, and particularly dependences between instructions. Thus, the question for speculative parallelization is: how predicting, with the best possible success rate, dependences that cannot be determined at compile time.

Any prediction process has to be based on some observations on how the code behaves, at least during a sample of its execution. Moreover, the observed behavior has to be used to feed the prediction mechanism. A representation model is required for this purpose. There is an important literature about prediction mechanisms, and many of these proposals have been, or could be, experimented to model the memory behav-

ior of a code. However, in the case of advanced speculative parallelizing and optimizing transformations, the model used has to provide enough information to guide code transformations, and also it must provide a way to build the required verification system, preventing dependences violations, and the associated rollback mechanism.

In this paper, we show a representation model allowing to handle advanced speculative transformations of loop nests. This model represents the sequences of memory addresses that are accessed by a loop nest as linear functions of loop indices. These functions are built by interpolating the address values referenced by each memory instructions during a short profiling phase.

3. THE LINEAR MODEL OF MEMORY BEHAVIOR

Speculative parallelization targets codes that cannot be analyzed precisely at compile-time. In the case of loops, *for*-loops whose bounds are linear functions of the enclosing loop indices, and whose memory accesses consists of accesses to array elements through linear reference functions of the loop indices can be analyzed at compile-time, i.e., statically. The well-known *polytope model* [Feautrier 1992a; 1992b; Bondhugula et al. 2008] is dedicated to such loops. In this model, dependence analysis is precisely performed and parallelizing transformations can be automatically generated [Bondhugula et al. 2008]. Loop nests that cannot be handled statically are characterized by memory accesses through pointers or array indirections. Loops with complex conditions and statically unknown iteration counts, i.e., *while*-loops, are also concerned. Such characteristics prevent to automatically parallelize these loops at compile-time. Even if more sophisticated memory dependence analysis can help, such as points-to-analysis [Chen et al. 2003], several memory accesses remain unresolvable at compile-time, making the parallelization fail.

Our proposal consists of applying the polytope model at runtime. First, the target loop nest execution is profiled during a short extract of its execution time, thanks to instrumentation instructions associated to each memory instruction, which collect the accessed memory addresses. Then, if possible, each collected address sequence is interpolated as a linear function of the loop nest indices. If successful, dependences are then computed from these linear functions. As it is done in the polytope model statically, these dependences are used to determine a "valid" parallel schedule of the loop nest statements and iterations, thus allowing to generate a parallelized version of the code, by applying linear transformation. Virtual loop indices are introduced in order to handle any kind of loops that might have complex conditions. Bounds of the inner loops are also interpolated as linear functions of the enclosing loop indices.

In the case of speculative parallelization, and since the dependences are computed from the observation of a short execution extract only, the term "valid" has to be translated to other properties. The computed dependences are used to predict that the same dependences will occur in the remaining iterations, and that the generated parallel code will not induce any access conflicts, and therefore no rollback will be required.

However, since the prediction can obviously fail, verifications have to be performed during the execution of the parallel code. With the linear model, this verification consists of comparing the actual accessed memory address to the values of the associated predicting linear function. If they differ, a rollback has to be performed.

Notice that memory accesses have to be performed in the parallel code in the same way as in the initial sequential code, i.e., using pointers or indirections. However, since the statements have been re-scheduled, these accesses have also to be modified in other to reference correct addresses. This is achieved by handling also scalar variables involved in the computation of the referenced addresses: their values also have to be interpolated as linear functions. If the accessed addresses can be represented as linear
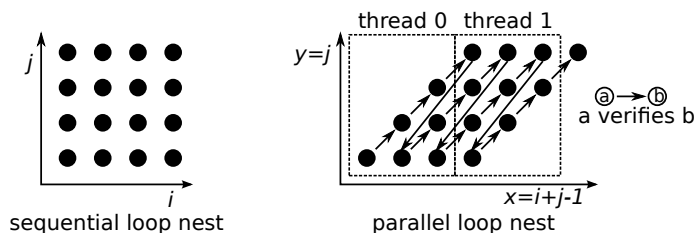
Fig. 2. Illustration of scalar values verification in the parallel loop nest

functions, there is a high probability for these related scalar variables to be represented as linear functions as well.

However, the need to handle these variables implies a strong constraint on the way the loop nest can be parallelized. The predicted scalar values must also be verified, while ensuring a correct computation of the accessed memory addresses. We propose the following restriction for the parallel schedules, that allows a relevant verification strategy. This restriction consists of avoiding parallel schedules which modify the order of the statements inside the bodies of the loops, or which induces loop fusion or fission. Only the order in which iterations are computed can be modified, and the parallel loop nest, whose outermost loop is parallelized, must keep the same global structure as the sequential loop nest. If so, a correct and efficient strategy is to initialize the scalar variables at the beginning of each iteration using their associated predicting linear functions. To verify that the variables modifications occurring in the loop body follow the prediction, their values are verified at the end of each iteration, by comparing them to the values of the linear functions used for their initializations in the next iteration according the sequential order. This strategy is illustrated in figure 2, where the initial iteration domain has been skewed.

Notice that iterations verified by other iterations that are executed by a different thread can be executed before being validated. Nevertheless, any dependence violation is necessarily detected at a given time, either before or after it occurs. However, when a violation is detected, the whole loop nest must be re-executed from start. This is obviously inefficient. Moreover, it does not allow to generate and run different parallel versions that could be more successful, due to the risk of costly rollbacks.

To overcome these problems, a solution is to consider contiguous chunks of the original loop, i.e., slices of the outermost loop, each of them being either parallelized or not. This chunking strategy also supports the run of small sequential instrumented chunks to profile the memory accesses and try to interpolate them. More generally, the size of the chunks can be dynamically adjusted depending on the stability of the program memory behavior. If a parallel chunk completes without being interrupted by the detection of a dependence violation, then a larger chunk, parallelized following the same schedule as the previous chunk, is launched. If a dependence violation is detected, then the chunk parallel execution has to be re-started until the point where the violation has been detected. Then an instrumented chunk, which is sequential, is launched. This chunk executes the violation point and a new linear modeling is attempted. If some accessed memory addresses and related scalar values cannot be represented by a linear function, a sequential chunk, whose size can also be adjusted, is launched. This mechanism is illustrated in figure 3.

The transactional memory system is based on memory backups circumscribed by the launched parallel chunks. Before launching a parallel chunk, the memory space which should be updated is predicted using the interpolating linear functions. Hence, by computing the minimum and maximum addresses of the updated memory locations,

Fig. 3. Illustration of the chunking mechanism

```
(1) initialize list with spaced chunks
******___******_******_____******____****** ...

(2) continue list with equally spaced elements
*___*___*___*___*___*___*___*___*___*___*-- ...

(3) continue with random spacing between elements
_**___*_*_*__*___**__**___***__**___****_** ...
```

Fig. 4. Memory allocation for the linked list

a memory copy is sufficient. In case of a dependence violation, the memory is restored from the backup and a new chunk processing again the restored data is launched. Notice that non-predicted memory locations cannot be modified thanks to the verification instructions.

4. EXPERIMENTAL RESULTS

We built a set of four benchmarks and simulated their execution as being handled by a framework implementing the proposed approach. Each benchmark is aimed to emphasize a particular characteristic. We generated the corresponding scenarios of sequences of instrumented, sequential and parallel chunks, by including any profiling, rollback and dependence analysis time penalties. Measurements were obtained by executing the benchmarks on 24 cores of two AMD Opteron 6172 800Mhz 12 core-processors running Linux 2.6.35.

4.1. Sparsely allocated linked list

This benchmark handles a large linked list allocated in memory in chunks and searches for an element of a given value. The memory areas reserved for the linked list are marked with '*'. They follow the pattern shown in Figure 4.

The first pattern may be recognized as a consequence of eliminating some elements from a list allocated contiguously. Parallelization can bring high benefits particularly when the processing of elements is computational intensive. Each of the chunks may be executed by a set of parallel threads, provided that the dependence analysis allows it.

The memory allocation exhibited by the second part of the list may appear when several data structures are allocated consecutively in the same loop. The elements, although not in contiguous memory areas but equally spaced, follow a new linear function, thus the chunk can be parallelized accordingly.
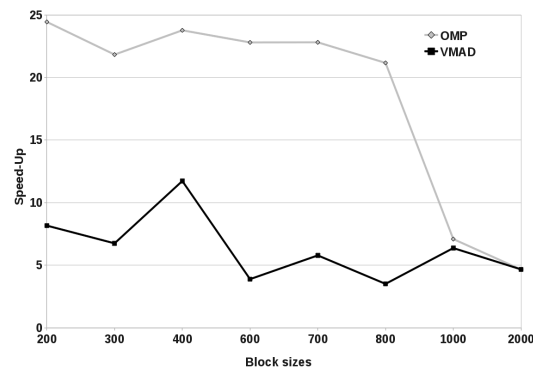
Fig. 5. Block-diagonal sparse matrix measurements

Finally, the third part of the list contains elements which are randomly distributed, possibly after many eliminations. This does not allow the parallelization of the chunk, nevertheless, it does not prohibit the parallelization of the previous chunks either.

The purpose of this example is to describe the mechanism of parallelizing by chunks, when the loop cannot be completely parallelized. In contrast to previous TLS approaches which would apply a rollback as soon as the memory access behavior changes, our approach is more adaptive and allows to alternate sequential and parallel chunks of the loop.

4.2. Block-diagonal sparse matrix multiplication

Matrix multiplication exhibits parallelism on multiple levels, yielding very good performance improvements. The same benefits can be obtained by parallelizing a block-diagonal sparse matrix multiplication. However, modern compilers cannot automatically detect this opportunity, due to accesses performed via indirections. As noted by Bruening et al in Softspec [Bruening et al. 2000], the Non-Hermitian Eigenvalue Problem Collection [Eig] contains numerous examples of sparse matrices with non-zero values organized in stripes or in blocks along the diagonal. Computations performed on such matrices are parallelizable, as each stripe or block displays a linear memory accessing behavior. By speculatively executing the loop in parallel, a rollback occurs in the first iteration of each new block. Nevertheless, the benefits of executing each block in parallel overcome this overhead. We carried out experiments with various block sizes and related our results both to the original sequential version, and to the parallel version obtained by inserting *omp pragma* in the source code. As expected, the results show that the width of the blocks has a direct impact on performance, the larger the blocks, the lower the number of rollbacks, hence the closer we get to the speed-up obtained by parallelizing the code by hand. The results are shown in Fig. 5.

4.3. Indirect References

The next benchmark initializes a two dimensional matrix B encoding the element occurrences of a banded matrix, linearized in a one-dimensional array A. These occurrences can be represented as a linear function of two variables. Next, the elements of array A are processed inside a nested loop of depth three, by using indirect references A[B[i][j]]. The code is shown in figure 6(a).

The profiling phase identifies that the elements of A are accessed by following a linear function, but dependence analysis shows that the loop cannot be parallelized in its original form, since each loop carries a dependence between consecutive itera-

tions. Therefore, we apply a polyhedral transformation that yields a dependence free outermost loop. The optimized code is displayed in figure 6(b). Not only we fully exploit the parallelism exhibited by the code, but we optimize the temporal locality of the accessed elements. In the sequential version, the element B[k][k] accesses the diagonal of matrix B, based on the index of the innermost loop, which yields very low spatial and temporal locality. Similarly for the element B[k][j], which accesses matrix B in column-major order. In the optimized version, as a consequence of the polyhedral transformation, element B[k][k] becomes B[z][z], depending on the index of the outermost loop, which results in very good temporal locality. Likewise for element B[z][y], depending on the outerloops z and y. Accesses to B[x-y][z] depend on the z index, which shows a good spatial locality.

```
for(i=0; i<N; i++)          for(z=0; z<=N-1; z++)
 for(j=0; j<N; j++)          for (y=0; y<=N-1; y++)
  for(k=0; k<N; k++)          for (x=y; x<=y+N-1; x++)
   A[B[i][k]] += A[B[k][j]]    A[B[x-y][z]] += A[B[z][y]]
               * A[B[k][k]];                + A[B[z][z]];
   (a) Sequential version       (b) Parallel version
```

Fig. 6. Indirect References to a banded matrix

We provided this example to demonstrate our strategy of applying polyhedral transformations at runtime, when speculative parallelization is not recommended otherwise.

4.4. Cherry Cake

Finally, to outline all main features of our approach in one example, we built the fourth benchmark which initializes a linked list and a two-dimensional matrix. It parses the linked list in a loop nest of depth two and processes the elements of matrix A differently, based on the values taken by the elements of the list. We built eight different cases, as illustrated in figure 7.

The list is initialized such that chunks of consecutive elements of the list have the same property and the memory accesses performed in each case follow a different linear function. Thus, the first *if* branch is executed for a number of consecutive iterations. The code corresponding to the first case displays a set of dependences that allow parallelization by applying a suitable polyhedral transformation. Next, the second *if* branch is executed and a rollback occurs, since the memory accessing behavior changed. We execute an instrumenting chunk to capture the new linear function. Next we parallelize again, by applying a new polyhedral transformation and we continue the strategy until the loop execution completes. In this example, our approach handles eight different parallelization phases for one single loop nest execution.

Overall, this benchmark emphasizes both the necessity of parallelizing by chunks and of applying different polyhedral transformations on each chunk.

We run it using four different chunking and rollback strategies in order to compare their efficiency. In each scenario, the profiling, sequential and parallel chunks have the default sizes 10, 100 and 100, respectively. Their characteristics are:

(1) When launching a chunk of the same type as the previous one, the chunk size is doubled. In case of a rollback, a sequential chunk of the default size is launched to re-execute all canceled iterations. Then a profiling chunk is launched.
(2) Similar to the previous strategy, the chunk size is doubled when the predicted behavior is unchanged. However, when a rollback occurs, we launch directly a profiling chunk. As a consequence, a parallel chunk of the same type as before might

```
curr = head;
for(i=0; i<N; i++)
 for(j=0; j<M; j++){

 if      (hasProperty1(curr->val))
    A[i+1][j] += curr->val + A[i][j];

 else if (hasProperty2(curr->val))
    A[i+1][j+1] += curr->val + A[i][j];

 else if (hasProperty3(curr->val))
    A[i+1][j] += curr->val + A[i][j+1];

  else if (hasProperty4(curr->val))
    A[i][j+1] += curr->val + A[i][j];

  else if (hasProperty5(curr->val))
    A[i][j] += curr->val + A[i][j+1];

  else if (hasProperty6(curr->val))
    A[i][j+1] += curr->val + A[i+1][j];

 else if (hasProperty7(curr->val))
    A[i][j] += curr->val + A[i+1][j];

 else if (hasProperty8(curr->val))
    A[i][j] += curr->val + A[i+1][j+1];

 curr = curr->next;
 }
```

Fig. 7.   Pseudo-code Cherry Cake

be executed, having the default size. If the first parallel chunk is invalidated, then a sequential chunk is launched, which will necessarily overcome the rollback point.

(3) The third strategy replaces the algorithm of doubling the chunk size, by a fixed size increment. This technique is aimed to test whether slower increases of the chunk size lead to lower rollback costs. As in the first strategy, in case of a rollback, a sequential chunk of default size is launched.

(4) Finally, the fourth strategy combines the method of an incremental chunk size and the technique of initiating a profiling as soon as a rollback occurs.

The speed-ups obtained by applying these strategies on the *cherry cake* example are shown in figure 8. This shows the advantage of incrementing the chunk size with a fixed value rather than doubling it, and ensures that the canceled chunks are smaller, which limits the overhead.

4.5. Results

Benchmark	Sequential exec. time (s)	Spec. par. exec. time (s)	Speed-Up
linked list	26.65	3.78	7.04
block-diag. matrix	5.4	0.4	11.72
banded matrix	219	8.4	26.07
cherry cake	516.23	57.15	9.03

Our benchmarks show considerable improvements in performance, as illustrated in the case of the *banded matrix* example. Super-ideal speed-up is obtained thanks to the polyhedral transformation which provides better data locality. Additionally, no roll-
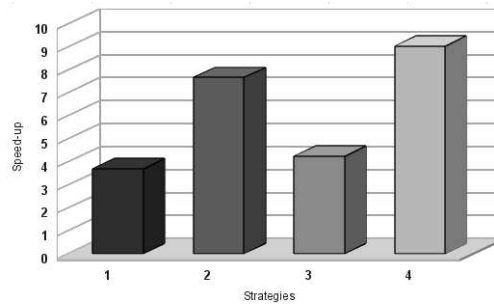
Fig. 8. Speed-ups with different chunking strategies

back is required, as the predicted behavior remains unchanged until the end of the execution. In the *linked list* and *cherry cake* examples, several changes in the memory accessing behavior are detected, which impose rollbacks and the generation of new parallel versions. This obviously has an impact on performance, however, it still outperforms the sequential executions.

5. RELATED WORK

There has been a considerable amount of research in TLS systems which propose speculative execution of threads, sometimes requiring architectural support. Since our proposal is a *software-only* framework, we review exclusively works of this type.

The previous TLS works can be classified into three main categories: proposals including compiler extensions, proposals handling code at runtime without any specific compilation phase, and proposals providing advanced code transformations. Most of these have a common feature: they speculatively parallelize loops by cutting their index range into contiguous parallel chunks.

In the first category, POSH [Liu et al. 2006] is a compilation framework for transforming the program code into a TLS compatible version, by using profile information to improve speculation choices. A similar approach is presented in [Johnson et al. 2007]. The Mitosis compiler [Quiñones et al. 2005] generates speculative threads as well as pre-computation slices (p-slices) dedicated to compute in advance values required for initiating the threads. The LRPD test [Rauchwerger and Padua 1995] speculatively parallelizes forall loops that access arrays and performs runtime detection of memory dependences. Such technique is applicable only when the array bounds are known at compile time. Tian *et al.* [Tian et al. 2008] focus on the efficient exploitation of pipeline parallelism using a data speculation runtime system which creates copies of statically, as well as dynamically allocated data, on-demand. Similar to [Raman et al. 2010], this study handles only single-level loops.

SPICE [Raman et al. 2008] is a technique using selective value prediction to convert loops into data parallel form. A similar approach is proposed in [Tian et al. 2010a]. In [Cintra and Llanos 2003], a speculative parallelization in chunks of the outermost loop is proposed, using a sliding window for reducing the impact of load imbalance. However this last proposal is limited to array-only applications.

Softspec [Bruening et al. 2000] is a technique whose concepts represent preliminary ideas of our approach. Linear memory accesses and scalar values sequences are detected, resembling our strategy, but only for innermost loops. Hence one-variable interpolating functions are built and used for simple dependence analysis via the *gcd* test. Thus, only the innermost loop can be parallelized. However, their initialization and verification mechanisms are similar to ours.

Finally, Zhong *et al.* present in [Zhong et al. 2008] several code transformation techniques to uncover the hidden parallelism and improve TLS efficiency, as speculative loop fission, infrequent dependence isolation or speculative prematerialization. However, these techniques do not attempt to generate several different parallel schedules.

Compared to previous works, although our approach requires phases of linear behavior of the code for parallelization, it applies advanced transformations for significantly improving the efficiency. Moreover, it does not require complex memory management between speculative and non-speculative data. Hence, it is orthogonal to some of the above proposals that provide efficient handling of non-linear behaviors. Merging our strategy with these techniques would yield good results for parallelizing codes alternating between linear and non-linear phases.

6. CONCLUSION

Thread-Level Speculation is a relevant technique for automatic and dynamic code parallelization. However, it suffers from strong limitations in the applicable parallelization strategies, yielding generally poor performance. In this paper, after having highlighted the main reasons of such limitations, we have shown that it is possible to go further in the parallelization strategies by proposing a linear modeling of the memory behavior allowing to apply the polytope model at runtime. We have shown on some synthetic benchmarks that this approach is effective.

REFERENCES

Non-Hermitian Eigenvalue Problem Collection .
http://math.nist.gov/MatrixMarket/data/NEP.

BONDHUGULA, U., HARTONO, A., RAMANUJAM, J., AND SADAYAPPAN, P. 2008. A practical automatic polyhedral parallelizer and locality optimizer. In *PLDI '08*. ACM, 101–113. http://pluto-compiler.sourceforge.net.

BRUENING, D., DEVABHAKTUNI, S., AND AMARASINGHE, S. 2000. Softspec: Software-based speculative parallelism. In *ACM Workshop on Feedback-Directed and Dynamic Optimization*. Monterey, California.

CHEN, D. K., TORRELLAS, J., AND YEW, P. C. 1994. An efficient algorithm for the run-time parallelization of doacross loops. In *Proceedings of the 1994 ACM/IEEE conference on Supercomputing*. Supercomputing '94. ACM, New York, NY, USA, 518–527.

CHEN, P.-S., HUNG, M.-Y., HWANG, Y.-S., JU, R. D.-C., AND LEE, J. K. 2003. Compiler support for speculative multithreading architecture with probabilistic points-to analysis. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '03. ACM, New York, NY, USA, 25–36.

CINTRA, M. AND LLANOS, D. R. 2003. Toward efficient and robust software speculative parallelization on multiprocessors. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '03. ACM, New York, NY, USA, 13–24.

FEAUTRIER, P. 1992a. Some efficient solutions to the affine scheduling problem, part 1 : one dimensional time. *International Journal of Parallel Programming 21,* 5, 313–348.

FEAUTRIER, P. 1992b. Some efficient solutions to the affine scheduling problem, part 2 : multidimensional time. *International Journal of Parallel Programming 21,* 6.

JOHNSON, T. A., EIGENMANN, R., AND VIJAYKUMAR, T. N. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '07. ACM, New York, NY, USA, 205–214.

LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., AND TORRELLAS, J. 2006. POSH: a TLS compiler that exploits program structure. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '06. ACM, New York, NY, USA, 158–167.

MICHAEL PHILIPPSEN, NIKOLAI TILLMANN, AND DANIEL BRINKERS. 2011. Double inspection for run-time loop parallelization. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2011)*.

OANCEA, C. E., MYCROFT, A., AND HARRIS, T. 2009. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. SPAA '09. ACM, New York, NY, USA, 223–232.

PATEL, D. AND RAUCHWERGER, L. 1999. Implementation issues of loop-level speculative run-time parallelization. In *Compiler Construction*, S. Jhnichen, Ed. Lecture Notes in Computer Science Series, vol. 1575. Springer Berlin / Heidelberg, 1–99.

PRABHU, M. K. AND OLUKOTUN, K. 2005. Exposing speculative thread parallelism in spec2000. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*. PPoPP '05. ACM, New York, NY, USA, 142–152.

QUIÑONES, C. G., MADRILES, C., SÁNCHEZ, J., MARCUELLO, P., GONZÁLEZ, A., AND TULLSEN, D. M. 2005. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '05. ACM, New York, NY, USA, 269–279.

RAMAN, A., KIM, H., MASON, T. R., JABLIN, T. B., AND AUGUST, D. I. 2010. Speculative parallelization using software multi-threaded transactions. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*. ASPLOS '10. ACM, New York, NY, USA, 65–76.

RAMAN, E., VA HHARAJANI, N., RANGAN, R., AND AUGUST, D. I. 2008. Spice: speculative parallel iteration chunk execution. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. CGO '08. ACM, New York, NY, USA, 175–184.

RAUCHWERGER, L. AND PADUA, D. 1995. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. PLDI '95. ACM, New York, NY, USA, 218–232.

TIAN, C., FENG, M., AND GUPTA, R. 2010a. Speculative parallelization using state separation and multiple value prediction. In *Proceedings of the 2010 international symposium on Memory management*. ISMM '10. ACM, New York, NY, USA, 63–72.

TIAN, C., FENG, M., AND GUPTA, R. 2010b. Supporting speculative parallelization in the presence of dynamic data structures. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. ACM, New York, NY, USA, 62–73.

TIAN, C., FENG, M., NAGARAJAN, V., AND GUPTA, R. 2008. Copy or discard execution model for speculative parallelization on multicores. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*. MICRO 41. IEEE Computer Society, Washington, DC, USA, 330–341.

ZHONG, H., MEHRARA, M., LIEBERMAN, S. A., AND MAHLKE, S. A. 2008. Uncovering hidden loop level parallelism in sequential applications. In *HPCA*. 290–301.

ZHU, C.-Q. AND YEW, P.-C. 1987. A scheme to enforce data dependence on large multiprocessor systems. *IEEE Trans. Softw. Eng. 13*, 726–739.