View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Dagstuhl Research Online Publication Server

Loop Tiling in the Presence of Exceptions

Abhilash Bhandari and V. Krishna Nandivada

Department of CSE, IIT Madras, Chennai, India
abilash@cse.iitm.ac.in, nvk@cse.iitm.ac.in

Abstract

Exceptions in OO languages provide a convenient mechanism to deal with anomalous situations. However, many of the loop optimization techniques cannot be applied in the presence of conditional `throw` statements in the body of the loop, owing to possible cross iteration control dependences. Compilers either ignore such `throw` statements and apply traditional loop optimizations (semantic non-preserving), or conservatively avoid invoking any of these optimizations altogether (inefficient). We define a loop optimization to be *exception-safe*, if the optimization can be applied even on (possibly) exception throwing loops, in a semantics preserving manner. In this paper, we present a generalized scheme to do exception-safe loop optimizations and present a scheme of optimized exception-safe loop tiling (oESLT), as a specialization thereof.

oESLT tiles the input loops, assuming that exceptions will never be thrown. To ensure the semantics preservation (in case an exception is thrown), oESLT generates code to rollback the updates done in the *advanced* iterations (iterations that the unoptimized code would not have executed, but executed speculatively by the oESLT generated code) and safely-execute the *delayed* iterations (ones that the unoptimized code would have executed, but not executed by the code generated by oESLT). For the rollback phase to work efficiently, oESLT identifies a minimal number of elements to backup and generates the necessary code. We implement oESLT, along with a naive scheme (nESLT, where we backup every element and do a full rollback and safe-execution in case an exception is thrown), in the Graphite framework of GCC 4.8. To help in this process, we define a new program region called ESCoPs (Extended Static Control Parts) that helps identify loops with multiple exit points and interface with the underlying polyhedral representation. We use the popular PolyBench suite to present a comparative evaluation of nESLT and oESLT against the unoptimized versions.

1998 ACM Subject Classification D.3.4 [Programming Languages] Processors – Optimization, Compilers

Keywords and phrases Compiler optimizations, semantics preservation, exceptions, loop-tiling

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.124

1 Introduction

Exceptions are one of the useful features in modern languages such as C++, Java, C#, ML and so on. Exceptions provide a structured way to handle anomalous and unexpected behaviors in the program and are finding increasing use in real world applications. While exceptions improve the programmability aspects, they have an impact on the generation of efficient code. The presence of exception throwing statements (explicitly in C++, Java, C#, ML – using a `throw` statement, or implicitly in C# and Java – for example, ArrayIndexOutOfBounds Exception) in the programs work as a deterrent to many compiler optimizations and analyses. This is because of the additional control flow edges and dependences (which cannot be resolved statically) that get introduced due to the presence of exceptions. We will illustrate the same using an example.

© Abhilash Bhandari and V. Krishna Nandivada;
licensed under Creative Commons License CC-BY

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 124–148

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
try{
 for(i=0;i<M;i++){
  for(j=0;j<N;j++){
    if(cond){
     throw e; }
    a[i][j]=b[j][i];
  }
 }
}catch(Ex e){ .. }
```

(a) Matrix transpose.

```
try{
 for(ii=0;ii<M/bn;ii++){
  for(jj=0;jj<N/bn;jj++){
   for(i=ii*bn;i<min(M,(ii+1)*bn);i++){
    for(j=jj*bn;j<min(N,(jj+1)*bn);j++){
      if(cond){throw e;}
      a[i][j]=b[j][i];
    } } } }
}catch(Ex e){ .. }
```

(b) An incorrectly tiled matrix transpose.

Figure 1 Matrix transpose and incorrect tiling.

Figure 1a shows the snippet of a code that computes the transpose of a matrix. The conditional `throw` statement (and the associated catch) is indicative of any statement that transfers the control (e.g., break, goto, return, or `throw` statement) from the body of the loop nest to an instruction outside the loop nest. The condition could be any of the possible sanity checks (e.g., array index bounds check) relevant in this context.

Figure 1b shows an incorrectly tiled loop [24], obtained by ignoring the control flow resulting from the conditional `throw` statement present in Figure 1a. The tiles are of size `bn*bn` and for simplicity, we assume that `M` and `N` are multiples of `bn`. It can be easily seen that the performed loop tiling is not semantics preserving (due to the control dependence). Similar reasoning can be given for many of the loop optimizations (e.g., software pipelining [24], loop interchange [24]), which makes them non-applicable, in the presence of exceptions.

Most of the compilers (e.g., GCC [32]) tend to be conservative and do not invoke many loop optimizations in the presence of exception throwing statements (inefficient). Some compilers (e.g., XLC [17]) provide command line switches to disable exceptions altogether and invoke the traditional optimizations (semantically non-preserving). There have also been many studies to identify unnecessary exception `throw` statements [8, 18, 7], and mark *exception-safe* regions [23]. While such a process can enable aggressive optimizations, it has its limitations owing to the specialized techniques used to target specific popular exceptions (for example, NullPointerException and ArrayIndexOutOfBoundsException) and the extent of information available at the time of compilation. To address these issues, we present a scheme to do *exception-safe* loop optimizations, especially when the optimizations may reorder the loop iterations. We define a loop optimization to be *exception-safe*, if it can be applied even on exception throwing loops, in a semantics preserving manner.

Though, in this paper for pedagogy, we use exceptions (as the control flow mechanism) and C++ (as the language of illustration), the presented concepts are equally applicable in the presence of other control flow statements (such as, goto, break, return) and programs written in a variety of high level languages that benefit from loop tiling.

Our Contributions

- A general scheme of backup, rollback (of the *advanced* iterations – ones that the unoptimized code would not have executed, if an exception is thrown) and safe-execution (of the *delayed* iterations – ones that the unoptimized code would have executed, but not executed by the optimized code) that can be used to derive the exception-safe variation of any existing loop optimization.

- Considering the importance of loop-tiling [24, 35], we specialize the exception-safe loop optimization scheme to derive an optimized exception-safe loop tiling scheme (oESLT – backs up minimal number of elements, and in case an exception is thrown, rolls back only the updates of the advanced iterations, and safely executes the delayed iterations). We also present a naive exception-safe loop tiling scheme (nESLT) that backs up every element and does a full rollback and safe-execution, in case an exception is thrown.
- We present a new program region called Extended SCoP (ESCoP), to help identify loops (with exception exits) that can be tiled.
- We implemented nESLT and oESLT in the Graphite framework of GCC 4.8. We present an evaluation over a set of base kernel benchmarks drawn from the popular PolyBench 3.2 benchmark suite [27]. We show that in the common case, when no exceptions are thrown, oESLT leads to significant performance gains (geometric mean 41.5%) compared to the base kernels (compiled using gcc -O3), at the cost of a minor memory overhead (geometric mean 0.3%). In this case, nESLT leads to similar gains in performance (geometric mean 40.8%), but incurs a much larger memory overhead (geometric mean 100%). If an exception is thrown, the impacts of oESLT and nESLT vary depending on the iteration in which the exception is thrown.

1.1 Related Work

Loop optimizations [24, 35] are arguably the most important optimizations implemented in the modern day compilers. Most of the recent advancements [2, 28, 29] in this space focus on improving the efficiency of the existing techniques. However, these works have mostly targeted programs that do not throw exceptions. The work of Yun et al [36] presents an optimal software pipelining scheme in the presence of simple control flow statements. In their scheme, software pipelining can be performed on code that includes control flows within the loop body. However, it still cannot handle loops that include exceptions or any other arbitrary jump statements that transfer the control out of the loop nest. Recently there has been increased interest [26] in designing optimizations for task parallel programs that may throw exceptions. Benabderahmane et al. [5] claim that the restrictions imposed by the polyhedral model are largely artificial and propose changes to the whole polyhedral optimization process such that the model can be more widely applied to the whole functions in a program. But it is not clear how their proposed scheme works in the presence of exceptions. To the best of our knowledge, ours is the first paper on exception-safety of traditional loop optimizations in general and exception-safe loop tiling in particular.

Analysis of programs that may throw exceptions has received a fair amount of interest, owing to the inherent scalability related issues therein. Sinha and Harrold [30] describe the effect of exception-handling constructs on the traditional control and data flow analyses and presents techniques to construct efficient representation for programs that may throw exceptions. Allen and Horwitz [1] use a similar representation to compute accurate slices, for programs that may throw exceptions. Choi et al [9] propose a compact representation of Control Flow Graph, called Factored Control Flow Graph (FCFG), for the exception related control flow in OO languages. Fu and Ryder [13] describe a static analysis technique that computes chains of semantically-related exception-flow links that can help improve the precision of control flow analysis. In contrast to these program analysis techniques, we present a technique to do exception-safe loop tiling, and the proposed analysis and transformation technique is not constrained by any scalability related issues.

Converting parts of programs that may throw exceptions to code that may not (e.g., by eliminating array out of bounds checks, null pointer checks and so on) is one of the

popular ways [7, 34, 8] to optimize programs that may throw exceptions. Loop versioning [22] is another promising approach where the compiler generates a specialized code with no exception `throw` statements; this specialized code is predicated with a series of checks that guarantee that no exception will be thrown. The main issue with these approaches is that they are specific to the pre-decided exceptions under consideration and it is quite challenging to extend the same to arbitrary conditional exceptions. In contrast, our proposed approach can handle any type of conditional `throw` statements, and we propose an extension to the base loop tiling optimization in the presence of exceptions.

Gupta et al [14] present a scheme to speculatively optimize the code, assuming that exceptions are never thrown. If an exception is thrown in the optimized code, they execute the unoptimized code (called the compensation code); this requires the complete backup of the updated array. In contrast, oESLT takes advantage of the underlying behavior of the loop tiling optimization to reduce the amount of backup, rollback and safe-execution.

There has been prior work on improving the scope and efficiency of speculative execution [4, 21, 10]. The main efficiency consideration here is that of minimal overhead and efficient recovery code generation. Another form of speculative execution with recovery is seen in the context of transactional memory [16, 12]. Our proposed method bears some resemblance to speculative execution, wherein we execute many tiles of the loop in a speculative manner and if an exception is thrown, we perform 'recovery'.

Song and Li [31] propose a scheme to tile loops speculatively, where the loop body may terminate prematurely because of convergence tests. In contrast to our proposed oESLT scheme, their scheme backs up all the array elements a priori (we only do partial backup), they backup the whole array many times, and their rollback requires writing of the whole array from the backed up store.

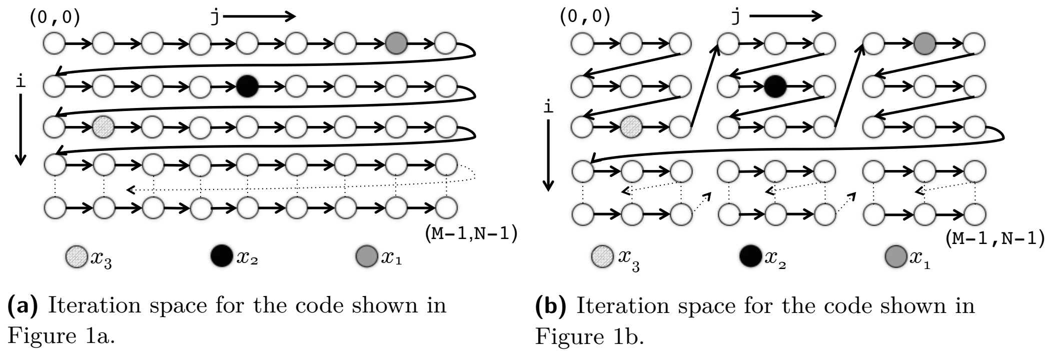
Our idea of backup and rollback has similarities to the work of versioning exceptions [25] that describes a language level extension, wherein the program can request a version of the store to be stored at a point of installation of a handler; when an exception is thrown, all the changes are reverted. Unlike our work, where for efficiency consideration the rollback is partial, the rollback in case of versioning exceptions is complete. It would be interesting to extend their work to handle arrays and partial rollback, and then using versioning exception to automatically generate rollback and safe-execution code, for loop tiling.

Outline: The rest of the paper is organized as follows. Section 2 presents a general scheme of exception-safe loop optimizations. Section 3 explains the general process of performing exception-safe loop tiling, along with a naive version thereof. Section 4 explains the scheme of optimized exception-safe loop tiling. We present the implementation and experimental results in Section 5 and conclude in Section 6.

2 Loop Optimizations in the Presence of Exceptions

In this section, we present techniques to do *exception-safe* loop-optimizations. Consider a normal loop nest L (consisting of loops with index variables $k_1, k_2, \ldots, k_n$) and a statement S present therein. Say, the free variables in S depend on a subset K of these index variables. Given a function $M : K \rightarrow Int$, we use $S(M)$ to represent the execution instances of S, where the index variable k_i gets the value $M(k_i)$, $\forall k_i \in K$. We now present some important concepts that form the basis of our exception-safe loop optimization scheme.

▶ **Definition 1.** Say the trace of the input loop consists of the sequence of statement instances $S(M_1), S(M_2), \cdots, S(M_n)$. If the same sequence forms the trace of the output loop, the transformation is said to be trace-preserving or else it is called trace-reordering.



(a) Iteration space for the code shown in Figure 1a.

(b) Iteration space for the code shown in Figure 1b.

Figure 2 Iteration space for the codes shown in Figure 1.

A trace-reordering transformation may result either via explicit reordering of the statements (example transformations: software-pipeline, instruction scheduling), or by reordering the iterations of a loop (example transformations: loop interchange, and loop tiling). Similarly, example trace-preserving transformations include loop peeling, loop unswitching and loop unrolling. The effect of exceptions on these two classes of loop transformations varies.

Trace-preserving transformations: Since the transformations do not change the sequence of instructions executed at runtime, such transformations can be oblivious to the presence of exception throwing instructions in the input program.

Trace-reordering transformations: The effect of exceptions on trace-reordering transformations is more involved. We will illustrate the same using the example.

Figure 2a shows the iteration space traversal for the code shown in Figure 1a. Each cell (`i,j`) represents an iteration. Consider the three iterations labeled x_1, x_2 and x_3. For the input program in Figure 1a, the order of execution of these iterations is $x_1 \prec x_2 \prec x_3$. The operator $\prec$ enforces an executes before relation.

Consider Figure 2b that shows the iteration space traversal for the code shown in Figure 1b (assuming, `bn`=3). The execution order for the previously considered iterations x_1, x_2, and x_3, in the transformed code is $x_3 \prec x_2 \prec x_1$. That is, compared to the iteration x_2, the iteration x_3 has been *advanced* and the iteration x_1 has been *delayed*. This difference in the execution order comes into limelight, when `cond` evaluates to `true` (say at iteration x_2). In Figure 2a, for an exception to be thrown in iteration x_2, the iteration x_1 must have been executed, and the iteration x_3 would not be executed after the exception is thrown. However, for the iteration space traversal shown in Figure 2b, for an exception to be thrown in iteration x_2, the iteration x_3 would have been executed, and the iteration x_1 would not be executed after the exception is thrown. Thus, the semantics of the transformed loop (Figure 1b) does not match that of the input loop (Figure 1a).

To make the code in Figure 1b *exception-safe*, if an exception is about to be thrown during the execution of the transformed loop (for example, at iteration x_2 in Figure 2b), the following additional steps need to be performed.

1. The execution of iterations that have been advanced (for example, iteration x_3) should be rolled-back (first rollback phase).
2. The delayed iterations (for example, x_1) should be executed in an order matching the traversal space shown in Figure 2a (safe-execution phase). If an exception is about to be thrown during any of these iterations (say, x_1) then roll-back the execution of all the iterations that have been advanced, with respect to x_1 (second rollback phase).
3. Throw the most recent exception object (from iteration x_1 or x_2).

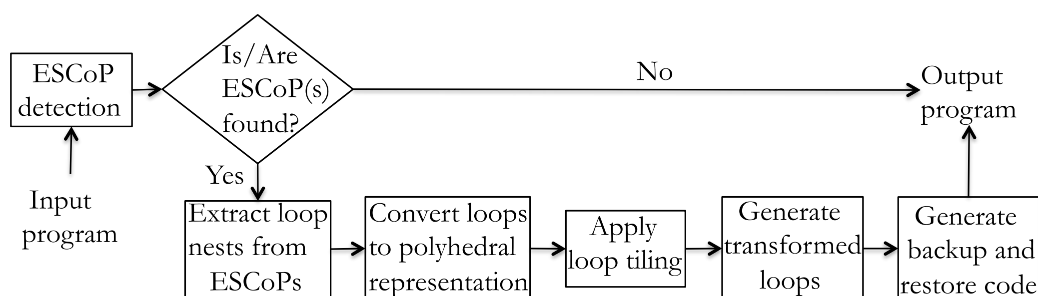


Figure 3 Exception safe loop tiling process.

These steps are applicable for any trace-reordering loop transformation. In this paper, we focus on loop tiling which is an important and popular trace-reordering transformation.

3 Exception safe loop tiling

Figure 3 shows the block diagram for our optimized exception safe loop tiling. We define a new type of program regions called Extended Static Control Parts (ESCoPs) (an extension of Static Control Parts (SCoPs) [5, 11]) that admit loops with possible exception related edges in the CFG. We then extract the loop nests from the ESCoPs and mark the control paths introduced by `throw` statements (*exception edges*). Next, we perform traditional loop tiling (blocking) on the extracted loop nests, which requires that there are no loop carried dependences (control or data) in the loop. Note: the loop carried control dependence introduced by the `throw` statements are masked by the ESCoPs. The tiled loop (for example, the code shown in Figure 1b) is then made *exception-safe* by emitting additional code to backup, to rollback and do safe-execution, if the input loop has exception edges.

Our proposed transformations handle only those loops which have affine bounds and affine conditions, but may contain `throw` statements. The conditions that bound the exception `throw` statements need not be affine as they are not modeled using the polyhedral representation. For the ease of explanation, we explain our techniques and algorithms over square tiling. However, the techniques are general enough to be extended to other shapes of tiles, provided the tiles are disjoint. We now explain the details of ESCoPs over which we implement our proposed algorithms and follow it up with a naive scheme of doing exception-safe loop tiling.

3.1 Extended SCoPs : Single Entry Multiple Exit Regions

A Static Control Part (SCoP) is defined as a region of consecutive statements comprising of loops with affine bounds and affine conditions, where the conditions depend only on either constants or loop invariant variables, or the loop index variables [5, 11]. The access functions used in a memory reference statement should be affine. The loops must be in canonical form.

An important feature of a SCoP is that all the loops that are a part of a SCoP have at most a single exit. Thus, a loop nest with multiple exits (arising due to a conditional `throw` statement present in the body of the inner most loop) is not part of a SCoP. Note, we need to ensure that the presence of exception edges do not inhibit the tiling of the input loop. To address these challenges, we define Extended SCoPs (ESCoPs) that admit SCoPs with a relaxation that the code therein may throw exceptions. The proposed ESCoPs have the following characteristics:

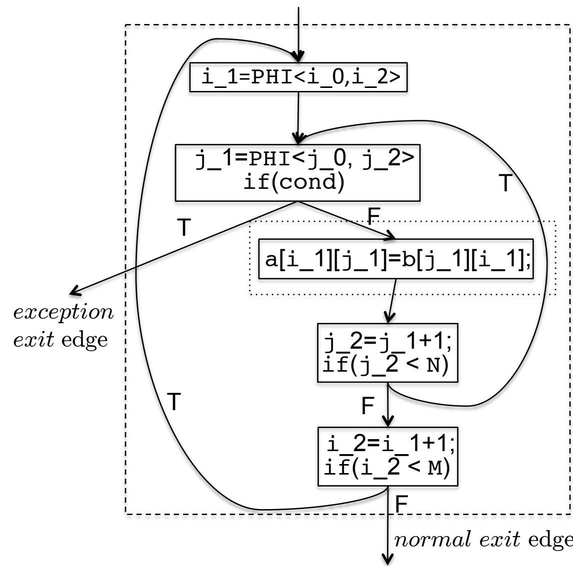


Figure 4 ESCoP and SCoP for the example code shown in Figure 1a.

1. ESCoPs are Single Entry Multiple Exit (SEME) regions.
2. Erasing all the throw statements from an ESCoP results in a SCoP – the underlying SCoP of the ESCoP.
3. The exit condition of the loop that is part of the underlying SCoP, of a given ESCoP, is called the *normal-exit condition* and the associated control flow edge is called the normal-exit edge. An ESCoP can have at most one normal-exit edge.
4. The exit conditions other than the normal-exit condition are termed *exception-exit* conditions (or *abnormal-exits* in GCC parlance), and the associated control flow edges are called the exception-exit edges.
5. Every loop belonging to an ESCoP need not have exception-exits. However, every loop belonging to an ESCoP should contain at most one normal-exit. Therefore, every SCoP is an ESCoP with zero exception-exits.

Figure 4 shows part of the control flow graph (CFG) for the loop nest shown in Figure 1a. The dotted region depicts the maximal SCoP detected, using the traditional SCoP detection algorithm [20], which unfortunately contains no loop. However, the corresponding ESCoP (depicted by a dashed box in Figure 4) includes the whole loop, along with the exception exit edge (the “T” labeled edge out of the second node). The “F” labeled edge out of the last node represents the normal-exit edge.

Given a program, we build its corresponding set of ESCoPs, such that each maximal loop nest present within a try-catch block or procedure boundary is associated with a unique ESCoP. Later, we perform exception-safe loop tiling on these ESCoPs. Our choice of ESCoPs is inspired from the fact that rollback and safe-execution operations have to be performed before the control is transferred to the exception handler. Thus, the rollback and safe-execution code can be inserted on the exception exit edges of the ESCoPs.

Note that in general, an ESCoP may contain multiple loops and some additional statements before/after the loops. For the ease of discussion, in this manuscript, we assume that each ESCoP has a single loop nest, which does not in anyway impact the generality of our proposed techniques.

```
for(i = 0; i < M; i++){ // backup phase
 for(j = 0; j < N; j++){
  bak[i][j] = a[i][j];
 } /* j */
} /* i */
try{
 try{ .. The loop nest of Figure 1b without the try-catch block ..
 }catch(...){ // all exceptions caught
   for(ti = 0; ti < (ii+1)*bn; ti++){ // rollback phase
    for(tj = 0; tj < N; tj++){
     a[ti][tj] = bak[ti][tj];
    } /* tj */
   } /* ti */

   for(i = 0; i < M; i++){ // safe-execution phase
    for(j = 0; j < N; j++){
     if(cond){ throw e; }
     a[i][j] = b[j][i]; } /* j */
   } /* i */
 } /* catch */
}catch(Ex e){ .. }
```

Figure 5 Impact of nESLT on the code shown in Figure 1a.

3.2 Naive Exception Safe Loop Tiling

The naive exception safe loop tiling (nESLT) approach speculatively tiles the loop assuming that no exceptions are thrown. It emits code (before the tiled loop) to back up all the elements that may be updated in the input loop (the backup phase). If an exception is thrown during the execution of the tiled loop then nESLT handles it in two phases. i) rollback: rolls back a set of elements that form an over-approximation of the actual updated elements, and ii) safe-execution: executes the untiled loop (from the beginning) till an exception is thrown. Note: if $\overrightarrow{I}$ and $\overrightarrow{J}$ are the iteration vectors when the exception is thrown in the tiled and untiled loop, respectively, then $\overrightarrow{I} \not\prec \overrightarrow{J}$ [24]. Further, if no exception is thrown, then the rollback and safe-execution phases are not invoked.

For the code shown in Figure 1a, Figure 5 shows the code generated by nESLT. The backup phase backs up all the elements of the array `a` into an array `bak`, and is used in the rollback phase, if an exception is thrown.

The main drawbacks of nESLT are that it conservatively does backup (all the elements, irrespective of when the exception is thrown), and rollback (all the updated elements and may be a few more) and safe-execution (more iterations than required). This can lead to a significant execution time and space overhead.

4 Optimized exception safe loop tiling

We now present our scheme of optimized exception-safe loop tiling (oESLT). We first explain some key ideas regarding backup, rollback, and safe-execution and then present the individual algorithms.

```
for(i=0;i<M;i++){
 for(j=0;j<N;j++){
  for(k=0;k<P;k++){
    if (cond) throw Ex;
    a[i][j]+= b[i][k]*c[k][j]; } } }
```

(a) complete-update-pattern

```
for(i=0;i<M;i++){
 for(j=0;j<N;j++){
   x[j] += b[i][j];
 }
}
```

(b) partial-update-pattern

Figure 6 Array update patterns.

4.1 Backup

Speculative execution of tiled loops may require certain updates to be rolled back and this in turn requires that the relevant older values are backed up. The exact values to back up depends on the array update pattern: *complete-update-pattern* (*cu-pattern*) or *partial-update-pattern* (*pu-pattern*).

cu-pattern: Consider the example code snippet shown in Figure 1a, where updates to a new element (for example, `a[i][j]`) start only after all the updates to the older elements (for example, `a[i][j-1]`) are complete. Such updates are termed as *complete-updates*. Figure 6a shows another example of code performing complete-update. At any iteration in this loop, there is at most one array element (the current element `a[i][j]`) that is partially updated. In case of complete-updates, all the distinct array elements updated in the previous iterations of the loop would never be updated again.

pu-pattern: In Figure 6b, each array element (for example, `x[j]`) is updated partially in each iteration of the outer loop. Such updates are termed as *partial-updates*. At any iteration in the loop, there are up to `N` array elements that are partially updated. If the updates to an array in a loop nest follow both cu-pattern and pu-pattern, we assume the array to be updated in the latter pattern only.

The goal of our optimized exception-safe loop tiling is to backup as few elements as possible and as few number of times as possible. For the sake of efficiency, we propose two different backup schemes: (i) Backup each element of the updated array, every time it is updated (backup for each iteration) – used for the array updates in input loop nest that follow the pu-pattern. (ii) Backup each element of the updated array exactly once (backup for each element) – used for the array updates in input loop nest that follow the cu-pattern.

4.1.1 Backup location and size

The location of insertion of the backup code and the size of the backup array also depend on the type of the array update pattern.

Case *pu-pattern*: Since we will use the first backup scheme (backup for each iteration), we backup just before every update and add the backup statements (copy) the innermost loop. Thus, the size of the backup array is bound by the size of the iteration space of the loop nest (e.g., the tiled version of the code shown in Figure 6b requires `M*N` space for backup).

Case *cu-pattern*: Even though the size of the updated array gives an upper limit on the size of the backup array, depending on the program point where the backup code is emitted (*backup point*), the actual size can be less. Interestingly, the backup point depends on the order in which the array elements (requiring backup) in the transformed loop are updated. The goal in this process (*efficiency consideration*) is to reduce the size of the backup array (reduces space overhead) and the number of elements backed up at one go,

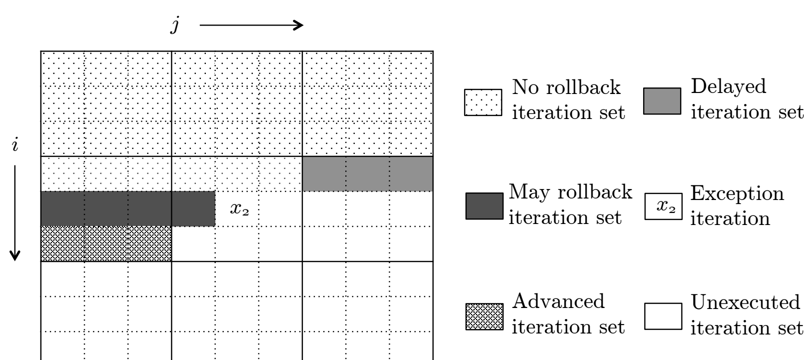


Figure 7 Advanced and delayed iterations.

```
for(ii=0;ii<M/bn;ii++){
 for(jj=0;jj<N/bn;jj++){
   /* Backup point. */

  for(kk=0;kk<P/bn;kk++){
   for(i=ii*bn;i<(ii+1)*bn;i++){
    for(j=jj*bn;j<(jj+1)*bn;j++){
     for(k=kk*bn;k<(kk+1)*bn;k++){
        if (cond) throw Ex;
        a[i][j]+=b[i][k]*c[k][j]; }}}
```

(a)

```
for(ii=0; ii<M/bn; ii++){
     /* Backup point */

 for(kk=0; kk<P/bn; kk++){
  for(i=ii*bn;i<(ii+1)*bn;i++){
   for(k=kk*bn;k<(kk+1)*bn;k++){
    for(jj=0; jj<N/bn; jj++){
     for(j=jj*bn;j<(jj+1)*bn;j++){
        if (cond) throw Ex;
        a[i][j]+=b[i][k]*c[k][j]; }}}
```

(b)

Figure 8 Two of the many possible tilings for the code shown in Figure 6a. For simplicity assume: `M`, `N` and `P` are multiples of `bn`. Backup code insertion point depends on the tiled code.

henceforth referred as *backup pulse* (reduces the execution time overhead, if an exception is thrown). We illustrate it with some examples.

To illustrate the possibility that the size of the backup array can be less than the size of the updated array, consider the tiled iteration space shown in Figure 7. We observe that for the first iteration of any row of tiles, the advanced iteration set and the delayed iteration set are empty. In other words, at the beginning of every row of tiles, the tiled loop execution is semantically equivalent to the unoptimized loop execution. Thus the advanced iteration set for any iteration contains the iterations only from the beginning of the current row of tiles and it is enough to backup these updates, and rollback in case an exception is thrown. This leads to a scenario where it is enough to allocate memory of size equaling that of a row of tiles, for the backup array. We now illustrate the importance of backup point, on the exact size of backup array and the backup pulse, with examples.

For the code shown in Figure 6a, Figure 8 shows two different tilings, and Figure 9 presents the backup codes thereof. Choosing any other program point in Figure 8 to emit the backup code would lead to increased overheads (space and execution time). For Figure 8a, inserting the backup code (within the loop with index `jj`) meets our specified efficiency consideration in the best possible manner (backup size = `bn*N`, backup pulse = `bn*bn`). Note that, it is enough to keep a backup of just one row of tiles at a time and for efficiency we can backup one tile at a time. Similarly, for Figure 8b, inserting the backup code (within

```
for(i=ii*bn;i<(ii+1)*bn;i++)
 for(j=jj*bn;j<(jj+1)*bn;j++)
  bak[i%bn][j]=a[i][j];
```

(a) Backup code for Figure 8a

```
for(i=ii*bn;i<(ii+1)*bn;i++)
  for(j=0;j<N;j++)
   bak[i%bn][j]=a[i][j];
```

(b) Backup code for Figure 8b

Figure 9 Backup codes.

the outermost loop with index `ii`) meets our specified efficiency consideration in the best possible manner (backup size = `bn*N`, backup pulse = `bn*N`). As another example, a tiling scheme that further exchanges the `ii` and `kk` loops of Figure 8b would lead to a scenario, where we have to backup the full array (size = `M*N`) all in one go (before the beginning of `kk` loop) – similar to nESLT.

4.1.2 Backup Algorithm

The backup algorithm is shown in Figure 10. The input loop nest $inpL$ (by ignoring the exception-exit edges) is transformed into trL using the traditional loop tiling technique. Every array update statement in the trL is handled separately and hence a unique backup array is used for every array that is updated in the loop.

For the input array update statement S, say R is its counterpart in trL. If the array update of S in $inpL$ follows the pu-pattern, the array element modified in S is backed up before each update. We use an auxiliary function 'emit' to generate code and || indicates a string concatenation operator. Note that the backup array, in this case, is indexed with the iteration vector of trL.

If the array update of S in $inpL$ follows the cu-pattern, the backup process is more involved. First the backup-point is determined and then the backup loop nest is generated. The variables $inpIdxSeq$ and $tiledGrpIndexSeq$ contain an ordered set of loop index variables. $inpIdxSeq$ contains the list of the loop-indices of $inpL$, corresponding to the loops that surround S; the index of the outermost loop comes first. For the code shown in Figure 6a, $inpIdxSeq$ = [`i, j, k`]. During the first phase of traditional tiling transformation, the loop nest is strip-mined [24]. For each loop in the input loop, the strip-mined loop has two loops: one loop (the outer one) iterates over groups of elements (*grouping loop*), and another loop (the inner one) iterates over all the elements in a given group (*element loop*). The variable $tiledGrpIndexSeq$ contains the list of loop indices of the grouping loops that surround R (for the code shown in Figure 8b, $tiledGrpIndexSeq$ = [`ii, kk, jj`]). We use two auxiliary functions getIndexSeq and getGroupIndexSeq to return the appropriate ordered sets. Given a loop index of a loop, the auxiliary function OriginalIndex returns the loop index of the input loop ($inpL$). For example, OriginalIndex function can be represented as a set of pairs {(`ii, i`), (`i, i`), (`jj, j`), (`j, j`), (`kk, k`), (`k, k`)}.

We emit the backup code just before that outermost grouping loop, at which the sequence $inpIdxSeq$ and OriginalIndex($tiledGrpIndexSeq$) do not match. This is the point after which the update pattern of the input loop differs from the transformed loop. The backup code is determined by the chop [19] computed for the update statement R, bound by the loop over the index variable tiledGrpIndexSeq[d]. For the two possible tilings shown in Figure 8a and Figure 8b, Figure 9 shows backup code to be emitted at the backup point specified. Note: Inserting the backup code on the critical path of execution may lead to i-cache and d-cache pollution. Therefore it is desirable to have a backup mechanism in which the backup execution does not come on the critical execution path of the input code. Hence in case of

```
1  Function GenBackupCode (inpL, trL, S, bak)
   Input: inpL: input loop nest that has been tiled; trL: transformed tiled loop nest; S:
          array update statement in inpL; bak the backup array.
2  begin
3    Say arr is the array variable updated in S and R is the counterpart of S in trL;
4    Say I⃗=index vector used to update the array in R;
5    updateType = type of array update in S;
6    if (updateType = cu-pattern) then
7      inpIdxSeq = getIndexSeq(inpL, S);
8      tiledGrpIndexSeq = getGroupIndexSeq(trL, R);
9      for (d = 0 ; ; d++) do // Compute the longest common prefix length.
10       if inpIdxSeq[d] ≠ OriginalIndex(tiledGrpIndexSeq[d]) then break;
11       ;
12     bakSlice = chop(R, getLoop(tiledGrpIndexSeq[d]));
13     emit bakSlice before the loop with loop index tiledGrpIndexSeq[d];
14     emit bak || "[" || I⃗ || "] =" || arr || "[" || I⃗ || "]" as the last statement in the
       body of the innermost loop of bakSlice;
15   else
16     Say I⃗1=iteration vector in which R is updated;
17     emit bak || "[" || I⃗1 || "] =" || arr ||"[" || I⃗ ||"]" before R;
```

Figure 10 Backup Algorithm.

"Backup for each element", the backup code is not inserted inside the innermost loop (though it is semantically correct to do so).

4.2 Rollback and Safe-execution

In this subsection, we present techniques to generate an efficient code to rollback and perform safe-execution. The main intuition behind this process is to rollback only the relevant computations (advanced iterations) and perform safe-execution of just the required iterations (delayed iterations).

4.2.1 Computing advanced and delayed iteration sets

Given an unoptimized loop, its tiled counterpart, and an iteration vector $\vec{p}$, we identify two sets of iterations:

1. $A(\vec{p})$: The set of iterations executed before $\vec{p}$ in the tiled loop.
2. $B(\vec{p})$: The set of iterations executed before $\vec{p}$ in the unoptimized loop.

If an exception is thrown in the iteration vector $\vec{p}$, then the set $(A(\vec{p}) - B(\vec{p}))$ consists of all the advanced iterations (with respect to $\vec{p}$) and hence the updates made by only these iterations need to be rolled-back. Similarly, the set $(B(\vec{p}) - A(\vec{p}))$ consists of all the delayed iterations and hence all these iterations have to be freshly executed (as part of safe-execution, may be in an untiled manner). As discussed earlier in this section, if an exception is thrown during this phase of safe-execution, say at iteration $\vec{q}$, then we have to further roll-back the execution of all the iterations that have been advanced with

respect to $\overrightarrow{q}$ (second rollback phase). The set of advanced iterations with respect to $\overrightarrow{q}$, after performing roll back for the iteration $\overrightarrow{p}$ is $((A(\overrightarrow{q}) - B(\overrightarrow{q})) - (A(\overrightarrow{p}) - B(\overrightarrow{p})))$. Note that $((B(\overrightarrow{q}) - A(\overrightarrow{q})) - (B(\overrightarrow{p}) - A(\overrightarrow{p}))) = \phi$, and hence there are no delayed iterations with respect to $\overrightarrow{q}$. As a result no further exceptions may be thrown and we do not need any further rollback phase. Note that if the unoptimized execution of the loop throws an exception, then the iteration at which the exception is thrown is given by $\overrightarrow{q}$ or $\overrightarrow{p}$, depending on whether any exception is thrown during safe-execution or not, respectively.

As an example, consider Figure 7 that shows the iteration space for the matrix transpose examples shown in Figure 1 (assuming `M=N=9` and a tile size of 3×3). Consider a specific iteration x_2. The set $A(\overrightarrow{x_2})$ is given by the iterations corresponding to the dotted, dark-grey and checked boxes. The set $B(\overrightarrow{x_2})$ is given by the iterations corresponding to the dotted, dark-grey and the light-grey boxes. Thus, if an exception is thrown at x_2, the checked-boxes $(= A(\overrightarrow{x_2}) - B(\overrightarrow{x_2}))$ correspond to the iterations that need to be rolled-back, and the iterations corresponding to the light-grey boxes $(= B(\overrightarrow{x_2}) - A(\overrightarrow{x_2}))$ should be safely executed.

For the loop in Figure 6a and its corresponding transformed code in Figure 8b (along with the backup code in Figure 9b), the loop with rollback code and safe-execute code is shown in Figure 11. For brevity we show the rollback and safe-execution loops for two of the total five interchanges (algorithms given below).

4.2.2 Generating efficient rollback and safe-execution code

An interesting challenge that was observed doing the generation of the above mentioned $A(\overrightarrow{p})$ and $B(\overrightarrow{p})$ sets is that these sets are non-convex in general. Performing the set-difference operations (in GCC) over non-convex sets to generate the sets corresponding to the advanced and delayed iterations, and the automatic generation of loop nests over such sets was (i) quite time consuming, and (ii) leading to inefficient code (owing to the complex convex set decomposition routines employed by the underlying library code). To overcome these challenges, we first identify the impact of loop tiling (in terms of reordered operations) as the union of the impact due to its constituent sub-transformations, and then generate the rollback and safe-execution code corresponding to each of these sub-transformations.

The tiling of a loop nest can be seen as strip-mining followed by a series of loop interchanges. The strip-mining transformation does not reorder the iteration space and only the loop interchange operations contribute to the final reordering. For the tilings shown in Figure 8a and Figure 8b, the sequence of interchanges are [(`j`, `kk`), (`i`, `jj`), (`i`, `kk`)] and [(`j`, `kk`), (`j`, `k`), (`jj`, `kk`), (`jj`, `k`), (`i`, `kk`)], respectively. An interchange (`p`, `q`) indicates that the outer-loop with index variable `p`, is interchanged with the inner-loop with index variable `q`.

For an iteration vector $\overrightarrow{p}$ over some iteration space, the advanced iterations set $A(\overrightarrow{p}) - B(\overrightarrow{p})$ and delayed iterations set $B(\overrightarrow{p}) - A(\overrightarrow{p})$ produced by a single interchange operation are convex sets. However, for an iteration vector $\overrightarrow{q} \in B(\overrightarrow{p}) - A(\overrightarrow{p})$ ($\overrightarrow{q}$ is an iteration of safe-execution phase in which an exception is thrown), then $((A(\overrightarrow{q}) - B(\overrightarrow{q})) - (A(\overrightarrow{p}) - B(\overrightarrow{p})))$ may not be a convex set. In case of the "Backup for each element" scheme, for simplicity, without loss of significant performance, we over-approximate this set to be the minimal enclosing convex set possible (max overhead bound by the tile size). In case of the "Backup for each iteration" scheme, we decompose the non-convex set into a series of convex sets.

4.2.3 Rollback and Safe-execution algorithm

Figure 12 shows the driver for generating the rollback and safe-execution code. The function InvokeRestorePhaseCode takes as input an auxiliary structure *trLStruct* that contains all

```
try{
  .. Code from Figure 8b, along with Figure 9b ..
}catch(...){ /* Rollback code start */
 for(p=i+1;p<(ii+1)*bn;p++){// Rollback for (i, kk) plane
  for(q=0;q<N/bn;q++){
   for(r=q*bn;r<min(N,(q+1)*bn);r++){
    a[p][r]=bak[p%bn][r]; } } }
 p=i; // Rollback for (jj, kk) plane
 for(q=jj+1;q<N/bn;q++){
  for(r=q*bn;r<min(N,(q+1)*bn);r++) {
   a[p][r]=bak[p%bn][r]; } }
               ··· Rollback for other planes. Not shown ...

 try{  /* Safe Execution code start */
  // Safe-execution for (i, kk) plane
  for(p=ii*bn;p<i;p++){
   for(q=0;q<N/bn;q++){
    for(r=q*bn;r<min(N,(q+1)*bn);r++){
     for(s=kk+1;s<P/bn;s++){
      for(t=s*bn;t<min(P,(s+1)*bn);t++){
       if (cond) throw Ex;
       a[p][r]+= b[p][t]*c[t][r]; } } } } }
  p=i;  // Safe-execution for (jj, kk) plane
  for(q=0;q<jj;q++){
   for(r=q*bn;q<min(N,(q+1)*bn);r++){
    for(s=kk+1;s<P/bn;s++){
     for(t=s*bn;t<min(P,(s+1)*bn);t++){
      if (cond) throw Ex;
      a[p][r]+= b[p][t]*c[t][r]; } } } }
           ··· Safe execution code for other planes. Not shown ...

 }catch(...){ /* Second rollback code start */
  for(x=p+1;x<=i;x++){ // Rollback for (i, kk) plane
   for(y=0;y<N/bn;y++){
    for(z=y*bn;z<min(N,(y+1)*bn);z++) {
     a[x][z]=bak[x%bn][z]; } } }
  x=i; // Rollback for (jj, kk) plane
  for(y=q+1;y<=jj;y++){
   for(z=y*bn;z<min(N,(y+1)*bn);z++) {
    a[x][z]=bak[x%bn][z]; } }
          ··· Second rollback for other planes. Not shown ...

  throw; // throw the last caught exception (second rollback code)
 } // end of second rollback and safe execution
 throw; // throw the last caught exception (first rollback code)
} // end of first rollback code.
```

Figure 11 Restore code.

```
1 Function InvokeRestorePhaseCode (trLStruct)
  Input: trLStruct: contains all the required information about tiled loop nest
2 begin
3   for (every exception-exit edge e in the tiled loop nest of trLStruct) do
4     e1 = GenRollbackCode(e, trLStruct); // Rollback phase
5     GenSafeExecutionCode(e1, trLStruct); // Safe-execution phase
```

Figure 12 Driver for rollback and safe-execution phases.

```
1 Function boundedSubtract (LoopB, LoopA, T, →I)
  Input: LoopB is the LoopNest on which a loop interchange operation is performed to
         obtain the loop nest LoopA; →I is an iteration vector; T is a statement in the
         body of LoopB and LoopA
2 begin
3   A=getDomain(LoopA, T, →I);
4   B=getDomain(LoopB, T, →I);
5   return subtract(A,B);
```

Figure 13 Parameterized bounds computation, followed by set subtraction.

the information about the tiled loop nest. For each exception edge in the tiled loop nest, the function InvokeRestorePhaseCode invokes the rollback phase code generator GenRollbackCode and the safe-execution phase code generator GenSafeExecutionCode. The rollback code is inserted at the destination of the exception edge. The exit point of the rollback code (returned by GenRollbackCode) is used by GenSafeExecutionCode to insert the safe-execution code.

Figures 13 and 14 show the sketch of two helper algorithms. The function boundedSubtract (Figure 13) takes as input two loop nests (derived by interchanging two loops therein), along with a common statement, and an iteration vector. The function getDomain ($LoopA$, T, $\overrightarrow{I}$) returns the set of iterations of $LoopA$ (executed before $\overrightarrow{I}$) in which T is evaluated. The boundedSubtract function returns the set-difference of these domains.

The function getRestoreStmt (Figure 14) takes an array update statement R as input and returns a statement T (that acts as the restore statement for R). The domain of T is set to the vector space of the index vector of R, if we use the "backup per element" scheme. Otherwise, it is set to the vector space of the iteration vector of the loop containing R.

The algorithm for the rollback is shown in Figure 15. Rollback loops are emitted to restore the computations performed in the advanced iterations. They are emitted for each interchanged plane in *interchanges*. The function getInterchanges returns the sequence of interchange operations performed on the input loop nest that produced the tiled loop nest trL. Each such interchange has an associated input loop and output loop.

A rollback loop is generated for each interchange operation. For each update statement in the body of the loop nest, we compute the restore statement (by invoking getRestoreStmt). We compute the iterations to rollback, by invoking *boundedSubtract* on the domains of trS and trL, and store in *itrsToRollBack*. We then project this set onto the domain of T, to compute the actual domain in which the rollback should happen. All such generated restore statements are fed to a loop generator (generateLoops) to generate the loop nest for the

```
1  Function getRestoreStmt (R)
   Input: R : an array update statement that is backed up
2  Say arr[I⃗] is the array updated in R;
3  if (backupScheme = "backup per iteration") then
4  |   Say T is the statement arr[I⃗] = bak[I⃗_1], where I⃗_1 is the iteration vector of the
   |   loop containing R;
5  |   domain(T)=vectorSpace(I⃗_1);
6  else // backup per element
7  |   Say T is the statement arr[I⃗] = bak[I⃗];
8  |   domain(T)=vectorSpace(I⃗);
9  return T;
```

Figure 14 A helper function to generate a restore statement.

```
1  Function GenRollbackCode (e, trLStruct)
   Input: e : entry edge to place the generated code; trLStruct: contains all the
          required information about tiled loop nest;
2  begin
3  |   interchanges = getInterchanges(trLStruct);
4  |   while (interchanges.hasNext()) do
5  |   |   op = interchanges.next();
6  |   |   trS = op.inputLoop(); trL = op.outputLoop();
7  |   |   for (every array update statement R in trS) do
8  |   |   |   T = getRestoreStmt(R);
9  |   |   |   p⃗ = iteration vector used to evaluate R in trS.
   |   |   |   itrsToRollBack=boundedSubtract(trS, trL, R, p⃗);
10 |   |   |   domain(T) = Project(itrsToRollBack, domain(T));
11 |   |   |   rollbackStmts.add(T);
12 |   |   ⟨exitEdge, newLStruct⟩ = generateLoops(e, rollbackStmts, trLStruct);
13 |   |   e = exitEdge ;
14 |   return exitEdge;
```

Figure 15 Rollback phase.

rollback phase. This function returns the new loop (*newLStruct*) and the normal exit point (*exitEdge*) of *newLStruct*. Figure 11 shows the rollback phase code for the two interchanges (`i`, `kk`) and (`jj`, `kk`).

The rollback loop generation phase is followed by the safe-execution loop generation phase. The safe-execution phase consists of two steps. In the first step, all the loop nests that execute the delayed iterations is generated. In the next step, a second set of rollback loops are generated which rollback the advanced iterations if an exception is thrown during the execution of the delayed iterations. We want to ensure that the delayed iterations are executed in the unoptimized order (so that if any exception is thrown during this execution, it matches the first exception that is thrown during the unoptimized execution). To aid in this process, we define a binary relation '$<_o$' over a pair of interchange operations. Say $op_a = (a_1, a_2)$ and $op_b = (b_1, b_2)$, we say that $op_a <_o op_b$, if a_1 is outer to b_1 (covariant) in

```
1  Function GenSafeExecutionCode (e, trLStruct)
   Input: e : entry edge to place the generated code, trLStruct: contains all the
          required information about the generated tiled loop nest.
2  begin
3  |  interchanges = getInterchanges(trLStruct).sort('<_o');
4  |  emit "try {";
5  |  while (interchanges.hasNext()) do
6  |  |  op = interchanges.next();
7  |  |  trS = op.inputLoop(); trL = op.outputLoop();
8  |  |  for (every statement S in trL) do
9  |  |  |  p⃗ = iteration vector used to evaluate S in trL.
   |  |  |  Domain(S)=boundedSubtract(trL, trS, S, p⃗);
10 |  |  |_ safeExecStmts.add(S);
   |  |  // generate the loop at edge e.  exitEdge represents the
   |  |     normal-exit edge
11 |  |  ⟨exitEdge, newLStruct⟩ = generateLoops(e, safeExecStmts, trLStruct);
12 |  |  op.setSafeExecLoop(newLStruct);
13 |  |_ e = exitEdge;
14 |  emit "} catch (Exception ex) {";
15 |  interchanges = getInterchanges(trLStruct);
16 |  while (interchanges.hasNext()) do
17 |  |  op = interchanges.next();
18 |  |  trS = op.inputLoop(); trL = op.outputLoop();
19 |  |  nlS = op.safeExecLoop();
   |  |  // Second Rollback phase.  Will not throw any exception.
20 |  |  for (every array update statement R in trL) do
21 |  |  |  T = getRestoreStmt(R);
22 |  |  |  p⃗ = iteration vector used to evaluate R in trL;
23 |  |  |  q⃗ = iteration vector used to evaluate R in nlS;
24 |  |  |  P=boundedSubtract(trL, trS, R, p⃗); // A(p⃗) − B(p⃗)
25 |  |  |  Q=boundedSubtract(trL, trS, R, q⃗); // A(q⃗) − B(q⃗)
26 |  |  |  Domain(T)=subtract(Q,P); // (A(q⃗) − B(q⃗)) − (A(p⃗) − B(p⃗))
27 |  |  |_ add(restoreStmtList, T);
28 |  |_ g = generateLoops(g, restoreStmtList, trLStruct);
29 |_ emit "} // end catch";
```

Figure 16 Safe Execution phase.

the input strip-mined loop, or $a_1 = b_1$ and a_2 is inner to b_2 (contravariant) in the input strip-mined loop. Given two interchange operations I_1 and I_2, if $I_1 <_o I_2$, then the execution of the delayed iterations resulting from the interchange operation I_1 precedes that of I_2, in the unoptimized order.

Figure 16 shows the algorithm to generate the safe-execution phase code. To generate the safe-execution code, we process the ordered loop-interchanges (sorted using the comparator '$<_o$'). For every statement (includes array update statements and conditional exception exits) in *trL*, the safe-execution phase domain is computed similar to that of the rollback

phase. The generated statements are passed to the function generateLoops to generate a new loop (stored in *newLStruct*). To handle any exception that may be thrown during the safe-execution code, we insert the safe execution code inside a try-block and generate the code for the handler (second rollback phase, see Section 4.2.1) in the corresponding catch-block. Figure 11 shows the code generated by the algorithm shown in Figure 16.

We highlight some of the interesting points about the code generation task: (i) The code for the rollback and safe-execute phases is added at the destination of the exception edge. This code is executed only when the abnormal-exit edge is taken, thereby leaving the i-cache unpolluted during the execution of the main loop. (ii) We generate the safe-execution loops in an order that ensures that given two interchanges I_1 and I_2 (say, $I_1 <_o I_2$), if an exception may be thrown in both the safe-execution loops generated for I_1 and I_2, at iteration vectors $\overrightarrow{p_1}$ and $\overrightarrow{p_2}$, then it can be guaranteed that $p_1 \prec p_2$. This is in accordance of our guarantee that we don't need any further rollback phase (see Section 4.2).

4.3 Discussion

Bounds: To explain the bounds on the number of backup elements, restore operations and safe-execution operations (for oESLT), we use Figure 1a as an example. We will assume that the tile is of size `bn*bn`.

The minimum size of the backup array is at least `bn*N` and the maximum size of the backup array is bound by the size of the array that is being updated (`M*N`), and this scenario occurs when the selected backup-point is outside the loop nest (oESLT, behaves like nESLT).

The minimum number of elements that need to be restored is 0 (for example, if an exception is thrown in the very first iteration). And the maximum number of elements that may be restored is `(bn-1)*(N-bn)`; for example, this case occurs, when the exception is thrown at the iteration `i=bn, j=N-bn`.

The minimum number of safe-execution operations performed is 0 (for example, if an exception is thrown at the iteration `i=bn, j=0`). The maximum number of safe-execution operations performed is bound by the maximum number of restore operations: `(bn-1)*(N-bn)`.

Handling exceptions and segmentation faults in backup code: The backup code may throw an exception (for example, NullPointerException or ArrayIndexOutOfBoundsException in languages like Java) or may lead to segmentation fault (in languages like C++) if the array to be backed up is uninitialized or if the array access is illegal. This issue can be addressed by performing the required checks on the updated array, before the execution of exception-safe tiled code. For example, consider the tiled code shown in Figure 8a and its backup code shown in Figure 9a. The check (`a!=NULL && size_a[0]>=M && size_a[1]>=N`), where `size_a[0]` represents the size of outer dimension (first dimension) of array `a`, is performed before the execution of the tiled code. The tiled code is executed, only if the check succeeds. Otherwise, the unoptimized code is executed.

Importance of ESLT: Even though exceptions may be thrown rarely, semantics preserving compilation requires that we cannot apply traditional optimizations (such as loop tiling) by ignoring exceptions. Even though our proposed semantics preserving optimizations do incur some minor overheads, we show in Section 5 that the resulting gains are significant.

5 Implementation and Evaluation

We have implemented our proposed techniques (nESLT and oESLT) in the Graphite framework of GCC-4.8. The Graphite framework provides support for polyhedral optimizations

Sl	Kernel name	Input Size
1	2mm(2)	4000 (EL)
2	3mm (3)	4000 (EL)
3	gemm (1)	4000 (EL)
4	syrk (1)	4000 (EL)
5	syr2k (1)	4000 (EL)
6	doitgen (1)	256 (L)

Figure 17 Benchmarks kernels and the input sizes. The numbers in the brackets indicate the number of tiled loops. EL = Extra Large, L = Large.

like loop tiling, loop interchange and so on, and is implemented as an optimization pass in GCC-4.8 (one of the 100s of passes). We use the powerful Integer Set Library (ISL) [33] of GCC compiler framework to implement the helper functions (discussed in Section 4.2) like 'boundedSubtract', 'Project', 'subtract', 'getDomain', and support representations like domain of a statement and iteration vectors. The 'generateLoops' function used in Section 4.2, uses the underlying Chunky Loop Generator framework (CLOOG) [3], to generate the transformed loops.

We have evaluated our techniques on the popular Polybench 3.2 (converted to C++) benchmark suite [27]. It is a collection of well known numerical and linear-algebraic kernels (containing loops) designed specifically for the study of different loop optimizations. The kernels can be run over a wide set of input sizes. Compared to application benchmark suites like SPEC [15] and PARSEC [6], Polybench (because of its small size and focussed loop kernels) has an advantage that it helps us localize the impact of the specific loop optimization under consideration, with little interference from the rest of the program. To evaluate the different overheads associated with our techniques, we need benchmarks where exceptions may be thrown in different computation loops (or in the absence of such benchmarks, we have to edit the loops in the existing benchmarks to throw exceptions conditionally). Considering the size and complexity of the benchmarks like SPEC or PARSEC, it is quite challenging to introduce such tunable conditional `throw` statements in all the loops therein. In contrast, because of the smaller sizes, the Polybench kernels make it quite easy to introduce such tunable conditional `throw` statements.

We converted all the thirty Polybench kernels to C++. Of these kernels, we found that only the six kernels shown in Figure 17 could be tiled and the rest could not be tiled (fourteen of the kernels had data dependences that could not be resolved, and another ten were found to be non profitable by GCC). To derive the base (unoptimized) versions of the six shown kernels, we performed a pass of loop-distribution (to generate perfectly nested loops), and perform function inlining of the computation kernel in the 'main' method (to satisfy some requirements of the underlying GCC compiler framework to perform tiling). We then added a conditional `throw` statement before the main computation statement, and nested each such loopnest inside a try-catch block. Figure 18 shows the sample transformations done in the 2mm kernel, to derive the corresponding base kernel. We introduce the surrounding try-catch block, so that the code after the loop nest (for example, code to print the execution statistics) can execute, even if the exception is thrown. Note that the conditional exception `throw` statements are added in the innermost loop before the array access statement so as to give an effect of some common `throw` statements like ArrayBoundsCheck, NullPointerCheck, and so on. We use these base versions as the input to our loop tiling phase. We chose the largest input (provided by PolyBench) such that the arrays can be allocated on the stack (a

```
main(){
  ...
  mm();
  ...
  print (execTime);
}

void mm(){
 for (i=0;i<NI;++i){
  for (j=0;j<NJ;++j){
   C[i][j] = 0; /* init */
   for (k=0;k<NK;++k){
    C[i][j]+=alpha*A[i][k]*B[k][j];
   }
  }
}
```

(a) Representative snippet of the original Polybench kernel 2mm.

```
main(){
 ...
 for (i=0;i<NI;i++){
  for (j=0;j<NJ;j++){
   C[i][j] = 0; /* init */ } }
 try{
  for (i=0;i<NI;i++){
   for (j=0;j<NJ;j++){
    for (k=0;k<NK;++k) {
     if (some-cond)
      throw 20;// throw some exception
     C[i][j]+=alpha*A[i][k]*B[k][j];
  } } } } catch (...) {}
 ...
 print (execTime);
}
```

(b) Modified 2mm kernel.

Figure 18 Typical transformations done on the PolyBench kernels to derive the base (unoptimized) versions.

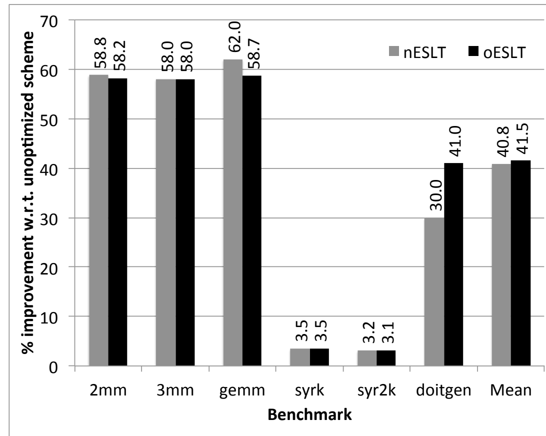
requirement of GCC loop-tiling). All the experiments were performed for square tiles. The size of the tiles was fixed by choosing the best tile size for the original benchmark kernels on our hardware: 16 for doitgen, and 20 for the rest.

All the experiments were performed on an Intel Xeon CPU E5-2670 system (with 32 KB L1 data and instruction caches, 256 KB L2 cache, and 20MB L3 cache). We present our evaluation over three different schemes: nESLT, oESLT, and the in built GCC loop tiling (here after termed as the unoptimized scheme). The unoptimized scheme does not tile any of these loops because of the presence of `throw` statements. Each of these schemes were invoked in GCC with -O3 -fgraphite options.

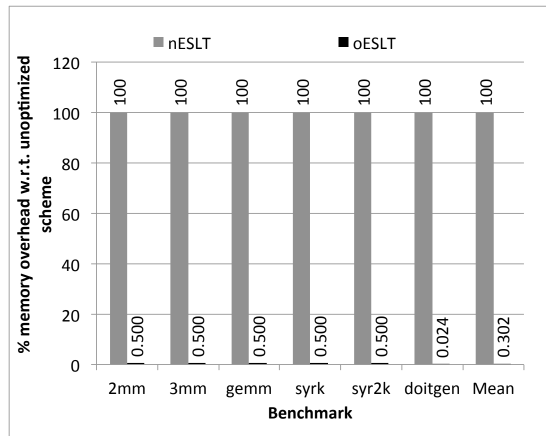
5.1 Impact of oESLT and nESLT when no exceptions are thrown

Figure 19 shows the impact of exception-safe loop tiling (ESLT) , when no exceptions are thrown inside the tiled loop; we measure the impact on both the execution time (Figure 19a) and memory (Figure 19b). We define percentage (%) improvement in execution time of scheme A over scheme B $= 100 \times (1 - \frac{\text{execution time with A}}{\text{execution time with B}})$. Similarly, we define percentage (%) overhead in the parameter under consideration (memory or execution time) of scheme A over scheme B $= 100 \times (\frac{\text{parameter value with A}}{\text{parameter value with B}} - 1)$. From Figure 19a, it can be seen that compared to the unoptimized scheme, on average oESLT and nESLT result in 41.5% and 40.8% improvements, respectively.

It can be seen that for most of the benchmarks (except doitgen) the impact of nESLT and oESLT on the execution time is similar; the maximum gap occurs for gemm, where the "backup all elements in one shot" scheme of nESLT helps in improved locality and leads to minor improvement. However, in case of doitgen, oESLT performs significantly better than nESLT. This is because, doitgen has a loop nest of depth 4, and during tiling, the outermost loop in the loop-nest is not tiled. Interestingly, the array update is based on the index of



(a) Percentage (%) improvement in execution time.



(b) Percentage (%) memory overhead.

Figure 19 Effect of ESLT when the nested conditional exception is never thrown.

the outermost three loop indices. As a result the number of elements in the backup pulse of oESLT is significantly small and hence the backup loop behaves as a prefetch loop, thereby improving performance.

Figure 19b shows the memory overhead incurred by both nESLT and oESLT, compared to the unoptimized scheme. It can be seen that the nESLT incurs significant memory overhead (geometric mean 100%). Compared to that oESLT reuses the backup space and reduces the memory overhead to a large extent (geometric mean 0.302%).

5.2 Overheads due to backup

To measure the overheads that we incur due to the insertion of backup code, we created a new ESLT scheme called the ideal ESLT (iESLT) scheme, wherein the loop is tiled, but has no backup code. Note: the conditional `throw` statements are retained (similar to oESLT and nESLT). In Figure 20, we compare the behavior oESLT and nESLT schemes against that of iESLT, when the input codes do not throw any exception. It can be seen that for most kernels the overhead due to the backup code is quite low (between -2% to 8.5%, geometric mean 1.39% for oESLT, and between 0% to 17%, geometric mean 2.69% for nESLT). In case

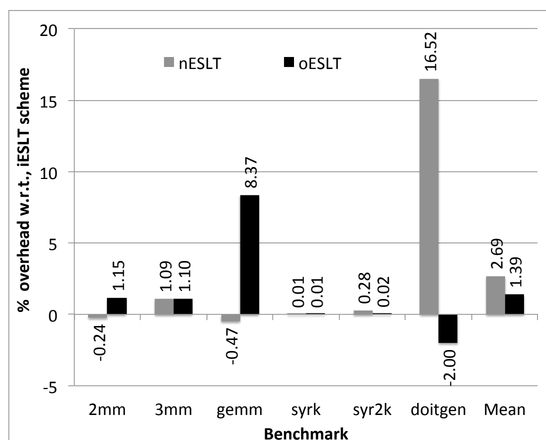


Figure 20 Percentage (%) overhead in execution time.

of doitgen, as discussed before, nESLT backups too many elements and thus incurs higher overhead. For the same kernel, in case of oESLT, interestingly, we realize slight performance gains because of the backup code! This we believe is because of the possible cache benefits due to the backup code (it acts as a prefetch loop). To establish this hypothesis, we increased the input size for doitgen (from 256 to 384 = 16*24) and found that the benefits due to the backup loop decreases; the backup loop stops working as a prefetch loop and hence pollutes the cache.

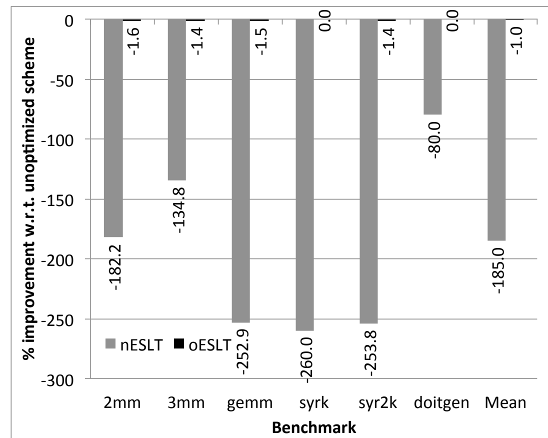
5.3 Impact of ESLT when exceptions are thrown

Figures 22 shows the impact of ESLT, when exceptions are thrown inside the tiled loop. Based on our experience with exceptions in OO programs that predominantly use exceptions to report corner cases (e.g., array being updated is null, the update to the array is not within the array bounds, and so on, which typically occur either at the beginning of the loop, or towards the end), we present a study of nESLT and oESLT, by forcing the exception to be thrown only in the first element of the first tile (Figure 21a), or last element of the last tile (Figure 21b) of the loop-nest.

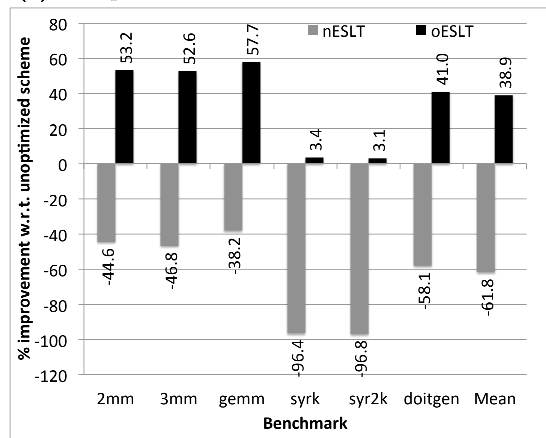
In Figure 21a, it can be seen that nESLT incurs significant performance degradation (ranging between 80.0% to 260.0%, geometric mean 185%) owing to the time spent in conservatively backing up all the array elements at the beginning. On the other hand, oESLT incurs a minor performance degradation (between 0.0% to 1.6%, geometric mean 1%) owing to the minimal backing up that is done as part of the optimized exception-safe loop tiling scheme.

In Figure 21b, it can be seen that nESLT again incurs significant performance degradation (between 38.2% to 96.8%, geometric mean 61.8%), owing to the time spent in conservative rollback and safe-execution of all the iterations. On the other hand, oESLT leads to significant improvements in performance (between 3.1% to 57.7%, geometric mean 38.9%). This is because the time savings resulting from tiling did offset the minimal time spent on rollback and safe execution.

Overall, it can be seen that oESLT leads to significant improvements when no exceptions are thrown (common case). In the not so common case, when an exception is indeed thrown, the gains depend on the exact iteration in which the exception is thrown (i.e., if the overheads are offset by the gains due to tiling).



(a) Exception thrown in the first tile.



(b) Exception thrown in the last tile.

Figure 22 Percentage (%) improvement in execution time.

Even though we use `throw` statements to represent the abnormal-exits, our proposed techniques can be used to handle any typical non-local control transfer statement (for example, goto, break, return, and so on). The techniques can also be applied on loops that contain multiple abnormal-exits of different types. Further, our techniques can handle conditional `throw` statements over both affine and non-affine conditions (a common scenario).

6 Conclusion and Future Work

In this paper, we present a generalized scheme to do exception-safe loop optimizations and present a scheme of optimized exception-safe loop tiling (oESLT), as a specialization thereof. oESLT leads to performance gains (because of loop tiling), with minimal overhead (due to backup and rollback).

Usually, the loops with exception `throw` statements (common in Java, C++) have multiple exit edges and to identify such loops in a general-purpose compiler (for example, GCC), we defined a new program region called ESCoP. We implemented our proposed techniques (built on top of ESCoPs) in GCC and show significant gains over the kernels from the PolyBench suite.

As a part of future work, we plan to understand the impact of the backup code on the tile size and the profitability. Designing an efficient exception-safe loop tiling scheme for parallel code is another interesting future work.

Acknowledgements. We would like to thank Shashidhar G for helping with the experimental setup, Rupesh Nasre and Raghesh Aloor for their insightful comments on a prior version of the manuscript and insightful discussions, in general. This research work is partially supported by the New Faculty Seed Grant, funded by IIT Madras (CSE/11-12/567/NFSC/NANV), the DRDO research grant (CSE/08-09/103/DRDO/HODX), and the DAE research grant (CSE/13-14/139/BRNS/NANV). We thank all these agencies for their generous funding and support.

References

1. M. Allen and S. Horwitz. Slicing java programs that throw and catch exceptions. In *PEPM*, pages 44–54, 2003.
2. R. Baghdadi, A. Cohen, S. Verdoolaege, and K. Trifunović. Improved loop tiling based on the removal of spurious false dependences. *TACO*, 9(4):52:1–52:26, 2013.
3. C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT*, pages 7–16, Juan-les-Pins, Sep 2004.
4. A. A. Belevantsev, S. S. Gaisaryan, and V. P. Ivannikov. Construction of speculative optimization algorithms. *Program. Comput. Softw.*, 34(3):138–153, 2008.
5. M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, LNCS, 2010.
6. C. Bienia, S. Kumar, J. Pal Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, pages 72–81, New York, NY, USA, 2008. ACM.
7. R. Bodík, R. Gupta, and V. Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
8. M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *JAVA*, pages 129–141, 1999.
9. J. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. *SE Notes*, 24(5):21–31, 1999.
10. J. Collard. Automatic parallelization of while-loops using speculative execution. *International Journal of Parallel Programming*, 23(2):191–219, 1995.
11. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20, 1991.
12. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.
13. C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in Java server applications. In *ICSE*, pages 230–239. IEEE, 2007.
14. M. Gupta, J-D Choi, and M. Hind. Optimizing Java programs in the presence of exceptions. In *ECOOP*, pages 422–446. Springer-Verlag, 2000.
15. J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, September 2006.
16. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300. ACM, 1993.
17. IBM. XL C/C++ Compiler. `http://www-03.ibm.com/software/products/en/xlcpp-aix`.

18 K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-in-time Compiler. In *JAVA*, pages 119–128, 1999.
19 D. Jackson and E. J. Rollins. Chopping: A generalization of slicing. Technical Report CMU-CS-94-169, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.
20 R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *PLDI*, pages 171–185, 1994.
21 J. Lin, W. Hsu, P. Yew, R. D. Ju, and T. Ngai. Recovery code generation for general speculative optimizations. *TACO*, 3(1):67–89, 2006.
22 V. V. Mikheev, S. A. Fedoseev, V. V. Sukharev, and N. V. Lipsky. Effective enhancement of loop versioning in java. In *CC*, pages 293–306, 2002.
23 J. E. Moreira, S. P. Midkiff, and M. Gupta. From flop to megaflops: Java for technical computing. *TOPLAS*, 22(2):265–295, 2000.
24 S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
25 V. K. Nandivada and S. Jagannathan. Dynamic state restoration using versioning exceptions. *HOSC*, 19(1):101–124, 2006.
26 V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar. A Transformation Framework for Optimizing Task-Parallel Programs. *TOPLAS*, 35(1):3:1–3:48, April 2013.
27 L. N. Pouchet. PolyBench: The Polyhedral Benchmark suite.
28 L. Renganarayanan, D. Kim, M. M. Strout, and S. Rajopadhye. Parameterized loop tiling. *TOPLAS*, 34(1):3:1–3:41, May 2012.
29 H. Rong, Z. Tang, R. Govindarajan, A. Douillet, and G. R. Gao. Single-dimension software pipelining for multidimensional loops. *TACO*, 4(1), March 2007.
30 S. Sinha and M. J. Harrold. Analysis and testing of programs with exception-handling constructs. *IEEE TSE*, 26(9):849–871, September 2000.
31 Y. Song and Z. Li. New tiling techniques to improve cache temporal locality. In *PLDI*, pages 215–228, New York, NY, USA, 1999. ACM.
32 R. M. Stallman and GCC DeveloperCommunity. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.8.0*. CreateSpace, Paramount, CA, 2013.
33 S. Verdoolaege. *ISL*: An integer set library for the polyhedral model. In *ICMS*, pages 299–302, 2010.
34 T. Würthinger, C. Wimmer, and H. Mössenböck. Array bounds check elimination in the context of deoptimization. *Sci. Comput. Program.*, 74(5-6):279–295, Mar 2009.
35 J. Xue. *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.
36 H. Yun, J. Kim, and S. Moon. Optimal software pipelining of loops with control flows. In *ICS*, pages 117–128. ACM, 2002.