IMPLEMENTATION AND EVALUATION OF REGISTER TILING FOR PERFECTLY NESTED LOOPS

A Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Master of Science in the
Graduate School of The Ohio State University

By

Bhargavi Rajaraman, M. Sc. (Tech)
Graduate Program in Computer Science and Engineering

The Ohio State University

2009

Master’s Examination Committee:
Prof. Sadayappan Ponuswamy, Advisor
Prof. Atanas Rountev
ABSTRACT

Tiling is a crucial loop transformation for generating high-performance code on modern architectures to expose coarse grain parallelism in multi-core architectures and to maximize data reuse in deep memory hierarchies. Register tiling improves Instruction Level Parallelism and is critical for these architectures to maximize performance improvements. Tiled loops with parameterized tile sizes facilitate runtime feedback used in iterative compilation and empirical tuning. Chunky Loop Generator (CLooG) is a powerful polyhedral code generator used to generate syntactic code from polyhedral representation of statement domains and data dependences. However, optimizations like loop unrolling and register tiling can only be applied syntactically. We implement register tiling algorithm for perfectly nested loops for rectangular and non-rectangular iteration spaces by post processing CLooG ASTs. There are numerous tools like Pluto, TLoG and HiTLoG that use CLooG to generate code after finding various optimizations through polyhedral approaches. An implementation of register tiling in CLooG can give higher performance improvements for generated code. Experimental results using a number of computational benchmarks comparing tiling techniques implemented in CLooG and TLoG, demonstrate the effectiveness of the implemented tiling algorithm.
ACKNOWLEDGMENTS

I am thankful to my advisor, Prof. P. Sadayappan for his guidance and support throughout the duration of M. S. study. I am thankful for all the efforts he took for my thesis. I would also like to thank Dr. Atanas Rountev for agreeing to serve on my Masters examination committee.

My heartfelt thanks to Muthu Baskaran, who was not only a great mentor but also a great friend. I am grateful to him for all his valuable discussions and support. I thank Albert Hartono, for his guidance and discussions.

I would also like to thank all my friends especially Karthik, Ketaki, Deepak, Jaidev, Tejas, Karthi, Siddharth and others who have made my stay at Ohio State a very memorable one.

I would like to thank my parents, brother Narayanan and cousin Mouli Raman who have been a great source of encouragement and inspiration.

I am extremely grateful to my husband Ravi, for all his love and support. He has been instrumental in motivating me to do my Masters and has been a source of great strength and support. Words cannot describe how much I owe him.

My thanks to God for all these people and for giving me this wonderful opportunity.
VITA

March 9, 1983 .......................... Born - Pondicherry, India

2004 ................................. M.Sc(Tech) Information Systems

2004 - 2005 ............................ Software Engineer, i2 Technologies Inc, Bangalore, India

2005 - 2007 ............................ Senior Software Engineer, Flextronics Software Systems, Chennai, India

2007-present ........................ Graduate Administrative Associate, The Ohio State University Libraries.

PUBLICATIONS

Research Publications


FIELDS OF STUDY

Major Field: Computer Science and Engineering
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Chapters:</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iii</td>
</tr>
<tr>
<td>Vita</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Importance of Maximizing Parallelism and Extracting Locality</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Importance of Loop Tiling</td>
<td>3</td>
</tr>
<tr>
<td>1.3 Introduction to Polyhedral Model</td>
<td>4</td>
</tr>
<tr>
<td>1.4 Organization of Thesis</td>
<td>5</td>
</tr>
<tr>
<td>2. POLYHEDRAL MODEL OF COMPILATION - AN OVERVIEW</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Polyhedral Representation</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Code Generation</td>
<td>9</td>
</tr>
<tr>
<td>3. CLOOG - CHUNKY LOOP GENERATOR</td>
<td>11</td>
</tr>
<tr>
<td>4. DESCRIPTION OF TILING ALGORITHM</td>
<td>16</td>
</tr>
<tr>
<td>4.1 Recent Work Related to Loop Tiling</td>
<td>17</td>
</tr>
<tr>
<td>4.2 Automatic Loop Tiling</td>
<td>18</td>
</tr>
</tbody>
</table>
4.2.1 Iteration Space Tiling ........................................ 20
4.3 Rectangular and Non-rectangular Iteration Space ............. 20
4.4 Approach to Register Tiling .................................... 22

5. EXPERIMENTAL EVALUATION OF CLOOG REGISTER TILING AND TLOG ............................................ 32

Appendices:

A. APPENDIX A CLooG AST DATA STRUCTURES AND FUNCTIONS 35
   A.1 Statement ....................................................... 35
      A.1.1 Root ...................................................... 36
      A.1.2 Assignment Statement ................................. 37
      A.1.3 Block Statement ........................................ 38
      A.1.4 For Statement ............................................ 38
      A.1.5 User Defined Statement ............................... 39
      A.1.6 Expression ................................................. 40
      A.1.7 Term Expression ......................................... 40
      A.1.8 Reduction Expression .................................. 41
      A.1.9 Binary Expression ....................................... 41
   A.2 CLAST Functions Description ................................ 42
      A.2.1 CLooG-CLAST Conversion function .................. 42
      A.2.2 Allocation and Initialization Functions ............. 42
      A.2.3 Memory Deallocation Functions ....................... 43
      A.2.4 CLAST Utility Functions added ....................... 43

B. IMPLEMENTATION OF REGISTER TILING IN ROSE ............ 45
   B.1 Rectangular Loop ............................................ 47
   B.2 Non-Rectangular Loop ....................................... 48

Bibliography ........................................................ 50
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Experimental Benchmarks</td>
<td>33</td>
</tr>
<tr>
<td>5.2 Best Code Execution Times (Time in milliseconds)</td>
<td>33</td>
</tr>
<tr>
<td>5.3 Best Reg-Tile sizes</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Polyhedral Model Example.</td>
<td>8</td>
</tr>
<tr>
<td>3.1 CLooG AST Statement Node - Types and Attributes.</td>
<td>14</td>
</tr>
<tr>
<td>3.2 CLooG AST Expression Node - Types and Attributes.</td>
<td>15</td>
</tr>
<tr>
<td>4.1 Steps in Automatic Tiling.</td>
<td>19</td>
</tr>
<tr>
<td>4.2 Example of Scalar Replacement.</td>
<td>21</td>
</tr>
<tr>
<td>4.3 Unrolling and Scalar Replacement.</td>
<td>21</td>
</tr>
<tr>
<td>4.4 Rectangular and Non-rectangular Tiled Iteration Spaces.</td>
<td>22</td>
</tr>
<tr>
<td>4.5 Example of Non-rectangular Iteration Space Tiles.</td>
<td>23</td>
</tr>
<tr>
<td>4.6 Tiled Code.</td>
<td>30</td>
</tr>
<tr>
<td>4.7 Tiled Code.</td>
<td>31</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

Optimizing compilers have become an essential part of modern high performance computer systems. Optimizing compiler frameworks apply various code transformations to reduce the execution time or code size of input programs. As emerging microprocessors offer very powerful parallel computing and deeper memory hierarchies, compiler transformations try to extract data locality and maximize parallelization to improve performance.

1.1 Importance of Maximizing Parallelism and Extracting Locality

Multi-core processors have become the norm of the day. Until early 2000s, increasing the clock frequency of microprocessors resulted in improved performance of most software as there were more instructions executed per clock cycle, resulting in lower execution times. Power dissipation issues, however, have rendered any further increase in clock frequency impossible. Increasing the number of cores on the chip has become the only solution to improve performance, while keeping power dissipation issues in check. Multi-core processors are used in all domains of computing - high
performance supercomputers built out of commodity processors, accelerators like the Cell processor and general-purpose GPUs and multi-core desktops.

With the ubiquity of multicore processors, there is a strong interest in technologies that can ease the development of parallel applications. Automatic parallelizing compilers start with sequential programs and automatically parallelize them, while preserving the semantics of the program. Loop transformations like loop tiling expose coarse-grain parallelism.

However, the increased number of threads (due to parallelization) results in increased memory bandwidth required by the program and it will be derogatory to program performance if we increase parallelization without regard to data locality as memory could become a bottleneck. Hence data locality and parallelization are two important goals for all compiler transformations.

Besides emergence of multi-core processors, disparity between processor and memory speeds has become sharper and caches or faster on-chip memories and registers are used to alleviate this difference in the speeds. Efficient use of cache memories is critical in determining program performance. Caches can be effective only if programs exhibit data locality, i.e. the property of programs to access elements that are located closer to each other in memory in succession.

Transformations like loop permutation, loop fusion and loop distribution aim to improve order of memory accesses to exploit all levels of memory hierarchy. Transformations like loop tiling rearrange the order of memory accesses so that elements in the cache are fully utilized. Transformations like unroll-and-jam and scalar replacement improve register reuse.
1.2 Importance of Loop Tiling

Loop tiling is a compiler transformation used by compilers to create block algorithms. The main benefit of loop tiling comes from the fact that within a tile, there is a high degree of data locality which algorithms can exploit, resulting in better cache or memory hierarchy performance. Also, if these tiles are independent of each other, different tiles can be allocated to different processors, thus exploiting the underlying parallel hardware. Thus loop tiling is a transformation that can exploit data locality and parallelism. Data reuse is important for good performance on modern high performance computers because the memory architecture for those machines is a hierarchy in which the cost of accessing data increases ten fold from one level of hierarchy to the next.

Loop tiling, when applied at the register level, increases ILP (Instruction Level Parallelism) and register reuse. In modern day processors, it is critical that performance at the register level be optimized, because modern microprocessors issue multiple instructions per cycle to exploit instruction level parallelism (ILP). To achieve high performance in these processors, compilers not only have to deal with data reuse but they must also extract ILP from the sequential version of the program to exploit the multi issue capabilities of the hardware. [12]

Loop tiling divides the iteration domain into tiles so that all the data referenced within a tile fits into the memory level. Even though implementation is the same for all levels of memory, register tiling involves the full unrolling of loops that execute within a tile (intra-tile loops). For rectangular iteration spaces, register tiling is the same as unroll-jam approach dealt in detail by [12] and is much simpler than for non rectangular iteration spaces, where fully unrolling the intra-tile loops is not trivial as,
number of times the intra-tile loops execute is not a constant and hence cannot be fully unrolled directly.

1.3 Introduction to Polyhedral Model

The polyhedral compiler framework is a model-based compiler framework that helps us to reason about transformations on loop nests with polyhedral representation for each statement in a loop nest and with inter- and intra- statement dependences represented by dependence polyhedra, totally discarding the syntactic representation of a program. The polyhedral representation can be readily applied to loop nests whose loop bounds and data access functions are affine. Program optimization in the polyhedral framework has three steps:

1. static dependence analysis of the input program.

2. transformations in the polyhedral abstraction.

3. generation of code for the transformed program.

In the last two decades, powerful compiler frameworks for dependence analysis and transformation of regular programs (programs with affine loop bounds and array access functions) using a polyhedral abstraction of statement domains and data dependences have been developed. CLooG is a powerful state-of-the-art code generator that is widely used to transform polyhedral representation of computation into loop structures.

In this thesis, we have implemented register tiling algorithm (unroll/jam) [8] for rectangular and non-rectangular iteration spaces for perfectly nested loops into CLooG by post-processing the abstract syntax tree (AST) of the loop structure generated by
CLooG. We have also done an experimental evaluation of register tiled code generated by CLooG versus TLOG.

1.4 Organization of Thesis

The rest of this thesis is organized as follows.

Chapter 2 will discuss the advantages of model based parallelism and gives a brief overview of the Polyhedral framework.

Chapter 3 gives a background of polyhedral code generation process and Chunky Loop Generator (CLooG)

Chapter 4 describes the register tiling algorithm implemented in CLooG

Chapter 5 describes the experimental set up and details of benchmarks run to compare tiling algorithm implemented in CLooG versus Tiled Loop Generator (TLOG)

Appendix A describes the CLooG Abstract Syntax Tree Data Structures and Functions.

Appendix ?? describes implementation of tiling algorithm in ROSE.
CHAPTER 2

POLYHEDRAL MODEL OF COMPILATION - AN OVERVIEW

In this chapter, we discuss the advantages of model based optimization methods as opposed to syntactic methods, and present a quick overview of polyhedral model of compilation. A more formal presentation of the model can be found in [10].

Many text based optimization methods textually modify the loops until some loops can be optimized and/or marked as parallel. As opposed to this, there is model based framework in which the textual structure of the source program is completely discarded and all transformations for locality and parallelism are reasoned based on a general mathematical model. The main benefit from this approach is:

- The right parallelization and optimization steps can be represented as one single transformation
- This transformation is discovered through classical mathematical optimization methods

The Polyhedral model for compiler optimization is a powerful mathematical framework based on parametric linear algebra and integer linear programming. Complex
execution-reordering, that can improve performance by parallelization as well as locality enhancement, is represented by affine transformations in the polyhedral model. In the last two decades, polyhedral model has evolved with development of powerful intermediate representation for performing transformations, and code generation methods (the process of generating source code from the intermediate representations after transformations).

In the polyhedral framework, there is a compact matrix representation of the dynamic instances of statements, and inter- and intra- statement dependences. The polyhedral compilation has the following three distinct steps:

1. Represent the input program in the polyhedral representation
   - Every loop in the program corresponds to a dimension in the polyhedral model.
   - Dependences between instances of statement are computed and represented as dependence vectors.

2. Apply a transformation to this representation

3. Generate the target (syntactic) code.

2.1 Polyhedral Representation

Polyhedral representation describes each statement separately with:

**Domain** - The domains are described using affine constraints that can be extracted from the program control (bound of enclosing loops)

**Schedule** An affine function assigning logical dates to iterations. A transformation is a set of affine scheduling functions. Each statement has its own scheduling
function which maps each run-time statement instance to a logical execution date.

Access functions - that capture array access semantics through parameterized affine inequalities. For any array reference, a given point in the iteration domain is mapped to an array element. An array access function is one that maps the iteration domain of any reference to array or scalar elements.

There are, however, restrictions on the structure of loop nests in the input program for which the polytope model can be applied. Parts of the program, called Static Control Parts can be represented algebraically in the polyhedral model. A SCoP is a
maximal set of consecutive instructions such that the only allowed surrounding control structures are for loops and if conditionals whose loop bounds and conditionals are affine functions of the surrounding loop iterators and global parameters.

Loops whose bounds are affine functions of loop iterators and conditionals that are affine functions of outer loop indices are statically predictable and are hence called static control parts. In such a program class, the iteration domain of every statement is a polyhedron. An iteration domain is defined by a set of equations where each equation is an affine function of domain dimensions. These static control programs or regular programs are readily accepted in the polyhedral model.

The key to polyhedral approach is that it clearly separates the four different types of program actions performed by program transformations.

1. Modification of iteration domain (loop bounds, strides) - loop unrolling, tiling
2. Modification of schedule of each individual statement - loop tiling
3. Modification of access functions (array subscripts) - privatization
4. Modification of data layout (array declarations) - padding, array merging (combining several arrays into a single one)

This kind of separation enables composition of transformations.

2.2 Code Generation

Once transformations have been applied in the polyhedral model, one needs to regenerate the target code, in the code generation phase. CLooG is a widely used code generator in the polyhedral model.
The code generation algorithm used in CLooG is based on the one developed by Quillere et al [9] and consists in a recursive application of domain projections and separations. The code generation algorithm recursively projects the statement polyhedra onto the outermost dimensions of the common embedded iteration space, separates the projections into disjoint polyhedra, generates loops that scan the disjoint polyhedra, and arranges the loops in the lexicographic order or according to the given affine scheduling function.
CHAPTER 3

CLOOG - CHUNKY LOOP GENERATOR

Code generation is the process of transforming a polyhedral representation into syntactic code. In the last two decades, there has been a significant progress in the development of powerful compiler frameworks for dependence analysis and transformation of regular programs, using a polyhedral abstraction of statement domains and data dependences [8]. CLooG is a state of the art widely used code generator which was originally developed to solve the code generation problem for optimizing compilers based on the polytope model. CLoog scans several polyhedra or unions of polyhedra at the same time, applying general transformations to the polyhedra and generates compilable code. It uses the best state-of-the-art code generation algorithm known as the Quillere et al. [9] algorithm with improvements and extensions [1]. CLooG is specifically designed to avoid control overhead and to produce very efficient code.

The constraints of both the domain and the context are what we will provide to CLooG as input. In CLooG, domains define the set of integral points to scan and their coordinates. In particular, CLooG has no knowledge about dependences on and between statements and hence on the legality of transformations. This dependence information has to be specified by the user and is usually done through the scattering
functions (to specify original schedule) which are used to force CLooG to scan in
a given order. Every kind of affine scanning order can be specified using scattering
functions. Each statement domain has its own scattering function and each scattering
function may be multi-dimensional. These schedules depend on the iterators in the
program and give each instance of each statement a unique logical execution date.
When scattering functions are not specified, scanning is performed in lexicographic
order of the original iterators. Thus any regular or static control program can be
specified using a set of iteration domains and scattering functions.

The key idea in the code generation algorithm is to recursively project the state-
ment polyhedra onto the outermost dimensions of the common embedded iteration
space, separate the projections into disjoint polyhedra, generate loops that scan the
disjoint polyhedra, and arranges the loops in the lexicographic order or according to
the given affine scheduling function. [8]

The register tiling algorithm explained in detail in the next chapter has been
implemented by CLooG by post processing on the CLAST data structures. Trans-
formations like unrolling and register tiling can only be done syntactically. CLooG is
a polyhedral code generator where it generates syntactic code from polyhedral rep-
resentations. Many tools like Pluto, TLoG and HiTLOG use CLooG to generate
optimized code. None of these tools have register tiling implemented in them. Hence
an implementation of parametric register tiling for perfectly nested loops can provide
good performance improvements for the generated code.

CLAST (Cloog Abstract Syntax Trees) is high-level syntactic AST representation
of the code, within CLooG. Figures (3.1) and (3.2) represent different data struc-
tures used by CLAST to represent the different AST nodes of a program. The solid
lines show the different types of an AST node. The dotted lines depict the different attributes of any given AST node.

CLAST is implemented in C and is available as part of the public distribution of CLooG.
Figure 3.1: CLooG AST Statement Node - Types and Attributes.
Figure 3.2: CLooG AST Expression Node - Types and Attributes.
CHAPTER 4

DESCRIPTION OF TILING ALGORITHM

Loop Tiling divides the iteration space into regular tiles or blocks of some shape and size and then traverses the tiles to cover the whole iteration space. Tiling changes the order of execution so that some outer loop iterations are executed before all inner loop iterations. Tiling reduces the number of intervening iterations between reuses. Therefore, reuses of data occur more closely in time and the amount of data fetched between data reuses is reduced. This allows reused data to still be in the cache and hence reduces memory accesses.

Tiling, when applied at the register level, is called register tiling. Even though the implementation of tiling is the same at all memory levels, register tiling involves fully unrolling the loops that traverse the register tiles. In non rectangular loop nests, fully unrolling the inner loop nests is not trivial [12] as the loop nests do not execute a constant number of iterations and hence unroll-and-jam cannot be applied directly. Register tiling improves ILP and register reuse.

Unroll and jam, like register tiling is a transformation that results in improving ILP and data locality. It involves unrolling outer loops and jamming resulting inner blocks together [13]. For rectangular iteration spaces, unroll-and-jam and register
tiling produce the same code, but they produce different code for non-rectangular iterations spaces.

### 4.1 Recent Work Related to Loop Tiling

Albert et al [8] have developed PrimeTile [4], a parameterized multi-level tile loop generator for arbitrarily nested imperfectly nested loops. The key idea used by them is to recursively decompose a loop at every level, into prolog, full-tile loop, epilog and cleanup loop. The prolog and epilog loops traverse non-rectangular boundary tiles, cleanup loop traverses rectangular boundary tiles and full-tiles traverse the full tiles and are register tiled. Dependence information is not being stored, and hence to enforce legality of tiling, the input code should first be made tileable by applying skewing and other unimodular transformations captures in the form of scattering functions or affine scheduling functions generated by Pluto.

TLOG, a parameterized tiled loop generator developed by Colorado State University has effectively addressed parametric single level tiling of perfectly nested loop nests by the polyhedral model. The TLOG algorithm decomposes the problem of generating tiled code into two sub-problems:

(i) scanning the tile origins, and

(ii) scanning the points within a tile.

Tiled code is generated by first generating a polyhedron that includes all tile origins and then scanning the polyhedron using CLooG. TLOG can only handle rectangular tiles, and hence it requires that the input program, if not originally rectangularly tileable, be transformed to make it rectangularly tileable[11]. Jimenez et al. [12]
addressed parametric tiled code generation for non-rectangular iteration spaces. The
code generated using their approach suffers from significant code expansion, but in-
volves lower overhead to scan through the full tiles in the code.

Bondhugula et al. developed Pluto [3, 14], a system for tiling arbitrary collections
of imperfectly nested loops. The Pluto system finds tiling transformations that result
in data locality optimization for sequential execution and communication minimiza-
tion for parallel execution. The Pluto system requires the tile sizes to be fixed for
code generation.

Transformations like unrolling and register tiling can only be done syntactically.
CLooG is a polyhedral code generator where it generates syntactic code from poly-
to generate optimized code. None of these tools have register tiling implemented in them. Hence an implementation of parametric register tiling for perfectly nested
loops in CLooG can provide good performance improvements for the code generated
by use of these tools.

4.2 Automatic Loop Tiling

The steps that are followed by a compiler while doing automatic multi-level tiling
are as follows:

**Dependence analysis**: determining whether the transformation is legal or not. With
perfectly nested loops, tiling is a valid transformation when a band of loops
is fully permutable. The condition for full permutability of a band of loops is
that all correspondingly permuted dependence vectors must be lexicographically
positive.
Locality analysis: determining the loops to be tiled at each memory level, the relative order of the loops, and the tile sizes that deliver the best performance.

Code updating phase: transforming the loop nest appropriately.

The code updating phase of multilevel tiling can in turn be classified into the following steps:

**Iteration space tiling phase** First, the iteration space is divided into different levels of tiles (one level for each memory level being exploited).

**Unrolling phase** The second step consists of fully unrolling the loops that traverse the iterations inside the register tiles.

**Scalar replacement** The last step consists in applying scalar replacement to expose reuse between iterations of the innermost loop.
4.2.1 Iteration Space Tiling

In the code updating phase, tiling is implemented by combining strip-mining and loop permutation [12]. Strip-mining splits a single loop into a pair of tile loop and element loop; the outer loop *tile loop* (also called the inter tile loop) steps between strips of consecutive iterations, and the inner *element loop* (also called the intra tile loop) traverses the iterations within a strip.

Loop permutation is used to establish the order in which the iterations inside the tiles are traversed. Loop permutation is applied so that in the final code, the inter-tile loops which step between tiles are the outer loops and intra-tile loops that traverse the points within a tile are the inner loops. Element loops traverse the register tiles and are fully unrolled in register tiling.

A loop is fully unrolled by replicating its loop body as many times as the loop bounds demand, changing the iteration variable that appears in the unrolled body by its different values.

Scalar Replacement is a transformation that uses dependence information to find out reused array values and replace it by scalar variables. Figure (4.3) shows example of scalar replacement.

4.3 Rectangular and Non-rectangular Iteration Space

Any iteration space has both boundary tiles and core tiles. A boundary tile is one whose intersection with the original iteration space is not equal to the full tile and a core-tile is one whose intersection with the original iteration space is equal to the full tile. When the element loops are traversing boundary-tiles, the number of iterations is less or equal than the tile size.
Figure 4.2: Example of Scalar Replacement.

(a) fully unrolling

\[
\begin{align*}
&\text{do } JJ = 1, N, B_{JJ} \\
&\quad \text{do } KK = 1, N, B_{KK} \\
&\quad \text{do } I = 1, N \\
&\quad \quad C(I,J,J,J) = C(I,J,J,J) + A(I, KK) \cdot B(KK,J,J,J) \\
&\quad \quad C(I,J,J,J+1) = C(I,J,J,J+1) + A(I, KK+1) \cdot B(KK+1,J,J,J+1) \\
&\quad \text{enddo}
\end{align*}
\]

(b) scalar replacement

\[
\begin{align*}
&\text{do } JJ = 1, N, B_{JJ} \\
&\quad \text{do } KK = 1, N, B_{KK} \\
&\quad \quad FF1 = B(KK,J,J,J,J) \\
&\quad \quad FF2 = B(KK+1,J,J,J,J) \\
&\quad \quad FF3 = B(KK,J,J,J,J+1) \\
&\quad \quad FF4 = B(KK+1,J,J,J,J+1) \\
&\quad \text{do } I = 1, N \\
&\quad \quad \quad C(I,J,J,J) = C(I,J,J,J) + A(I, KK) \cdot FF1 \\
&\quad \quad \quad C(I,J,J,J+1) = C(I,J,J,J+1) + A(I, KK+1) \cdot FF2 \\
&\quad \quad \quad C(I,J,J,J+1) = C(I,J,J,J+1) + A(I, KK+1) \cdot FF3 \\
&\quad \quad \text{enddo}
\end{align*}
\]

Figure 4.3: Unrolling and Scalar Replacement.
Figure 4.4: Rectangular and Non-rectangular Tiled Iteration Spaces.

Difference between a rectangular and non-rectangular iteration space is that, in rectangular iteration space, the boundary tiles are rectangles and hence element loops execute a constant number of iterations. Whereas in non-rectangular iteration spaces, since an inner loop bounds are arbitrary functions of outer loop iterators, the element loops do not execute a constant number of iterations. It is precisely this property that makes register tiling for non-rectangular iteration space non trivial since a condition for fully unrolling is that the loops must execute a constant number of iterations.

4.4 Approach to Register Tiling

We implement the algorithm originally implemented in Orio [7] for register tiling. We have implemented the same algorithm for register tiling in CLooG. Orio is an empirical annotation system that performs a number of optimizations (like tiling, register tiling, scalar replacement, loop fission, loop fusion) on a specified code fragment and performs an empirical search of selecting the best among optimized code
Figure 4.5: Example of Non-rectangular Iteration Space Tiles.
variants. PrimeTile [4], is an enhancement to Orio, which is capable of handling register handling for arbitrary imperfectly nested, parameterized loop nests.

We need to enforce legality of tiling since CLooG does not store any dependence information. The input code should first be made tileable by applying skewing and other unimodular transformations captured in the form of scattering functions or affine scheduling functions generated by Pluto. These scattering functions are also specified in the input file given to CLooG, thus ensuring that program semantics are preserved while applying register tiling.

Broadly, index set splitting is used to isolate a partition where the loops iterate exactly as many times as the tile size in their dimension. These loops generate the full tiles along that dimension. In this particular partition the loops are tiled and intra-tile loops are fully unrolled. In the other partitions, we retain the original statement without tiling/unrolling. Index set splitting is a code transformation that splits a loop into two new loops, where each new loop iterates over non-intersecting partitions of the original loop. ISS does not change the order in which individual iterations are executed. Therefore, it always is a legal transformation.

The algorithm recursively decomposes every loop into prolog loop, full-tile loop, epilog loop and cleanup loop. A depth first search on the AST is performed, and algorithm is recursively applied at every level of the loop nest.

The algorithm, for a given loop nest, hence can be summarized as following steps.

1. Index-set split the loop nest into main loop and cleanup loop

   - Cleanup loop covers rectangular boundary tiles

2. Partition the main loop into prolog, full-tile and epilog loop.
• To identify full tile loops, we explicitly scan to find loop bounds for full-tile loops.

• Full-tile loops cover core tiles, whereas prolog and epilog loops cover boundary non rectangular tiles

3. Fully unroll full-tile loop identified in step 2.

In rectangular iteration spaces, it is very easy to break the tiled loop nest since the intersection of a boundary-tile with the original iteration space is always a rectangular space and, therefore, it is enough if we just adjust the bounds of the outer tile loops. In this case, if we want to tile the outer i loop, first we do an index set splitting on the i-loop. We can split the loop into two segments the first segment, called the full-tile loop which we will further tile, and the second segment, which is left untiled and we call the clean-up loop, covers all the remaining iterations from $i = it, i \leq ubi$.

As an example, we use a 1-D loop nest shown below:

```plaintext
for i=lb_i, i<ub_i, i+=st_i
S(i,j)
```

After index-set splitting, for a tile size of $T_i$ for the i-loop, we get the following loop segments:

```plaintext
for it=lb_i, it<=ub_i-T_i+st_i, it+=st_i
S(i,j)
```

```plaintext
for i=it, i<ub_i, i+=st_i
S(i,j)
```
The second step is the tiling step. We tile the main loop segment (obtained from
the index set splitting method) into an outer, inter-tile loop which generates all tile
origins and the inner intra-tile element loop that covers points within a tile. For
rectangular iteration space, main loop tiling generates only full tiles.

\[
\text{for } it=lbi, it<= ubi-Ti+sti, it+=Ti \\
\text{for } i=it, i<=it+Ti-sti, i+=sti \\
\text{ } \\
S(i) \\
\text{for } i=it, i<ubi, i+=sti \\
\text{ } \\
S(i) \\
\]

The third step is to fully unroll the intra-tile loops of the main loop. This is
easy in rectangular iteration spaces, since in full tiles, element loops always execute
a constant number of iterations.

\[
\text{for } it=1bi, it<= ubi-Ti+sti, it+=Ti \\
\text{ } \\
S(it) \\
S(it+1) \\
S(it+2) \\
. \\
. \\
. \\
S(it+sti) \\
\text{for } i=it, i<ubi, i+=sti \\
\text{ } \\
S(i) \\
\]
Thus, for rectangular iteration space, no prolog and epilog loops are generated but simply full-tile and cleanup loops. In the case of non rectangular iteration space, prolog, full-tile, epilog and cleanup loops could be generated. Epilog and prolog loops traverse non rectangular boundary tiles whereas cleanup loops traverse rectangular boundary tiles.

In nonrectangular iteration spaces, the first index set splitting step is the same as the one in rectangular iteration space and it splits the loop into main loop and cleanup loop. Then, while tiling the main loop segment, the key idea in register tiling is to separate full tiles and partial tiles. Every loop is separated into three parts - the prolog loop, full tiles, and epilog loop. Prolog and epilog loop nests cover partial tiles. Full tiles are covered by loops whose lower and upper bounds are respectively \( it \) and \( it + Ti \), whereas partial tiles are those whose upper bound is not \( it + Ti \) (since partial or boundary tiles do not cover a full tile).

Let us consider the simple 2D perfectly nested, non rectangular loop shown here:

\[
\text{for } i=lb_i, ub_i, st_i \\
\quad \text{for } j=lb_j(i), ub_j(i), st_j \\
\quad S(i, j)
\]

The perfect loop nest contains an inner loop \( j \) whose bounds are arbitrary functions of the outer loop variable \( i \). Consider a non-rectangular iteration space displayed in Figure (4.5), corresponding to the perfect loop nest in this example. Since, \( i \) is the outermost loop, tiling \( i \) is straightforward (rectangular space tiling). We first do index-set splitting on \( i \) as well as tile \( it \), thus generating as many full tiles along \( i \) as possible. The following figure shows the corresponding code structure, with the first
loop covering all full tiles along $i$, followed by a clean-up loop segment that covers
the remainder of iterations that did not fit in a complete $i$-tile of size $T_i$.

/* Full-tile loop along $i$ */
for $it=lbi$, $it<=ubi-Ti+sti$, $it+=Ti$
   for $i=it$, $i<=it+Tisti$, $i+=sti$
      /* Code tiled along $j$ dimension */

/* Cleanup-loop along $i$ */
for $i=it$, $i<ubi$, $i+=sti$
   for $j=lb(i)$, $j<=ub(i)$, $j+=stj$
      $S(i,j)$

For each $i$-tile, full tiles along $j$ are identified. We have implemented explicit, run-
time scanning to find start of full tiles. In contrast, PrimeTile implements static loop
bound calculation. The essential idea is that the largest value for the $j$-lower bound
(newlbj) is determined over the entire range of an $i$-tile and it represents the earliest
possible $j$ value for the start of a full tile. Similarly, by evaluating the upper-bound
expressions of the $j$ loop, the highest possible $j$ value (newubj) for the end of a full
tile is found. If newlbj is greater than newubj, no full tiles exist over this $i$-tile range.
If newlbj is less than newubj, the executed code has three parts:

**a prolog** for $j$ values from $lb(j)$ upto newlbj - stj (where stj is the loop stride in $j$
dimension)

**a full-tile segment** from $jt$ upto $jt+Tj-stj$

**an epilog** for $j$-values greater or equal to $jt$ and upto $ub(j)$

28
The code for the full-tile segment is generated using a recursive procedure that traverses the levels of nesting. The detailed tiled code for this example is shown below:

In the above code, the unrolled-and-jammed loop body is represented as unrolled-and-jammed(i,j) with $T_i \times T_j$, which corresponds to the following expanded form:
/* full-tile loop for iterator i */
for it=lb_i, ub_i-T_i+st_i, T_i
  [ new_lb_j = -\infty
  new_ub_j = \infty
  for i=it, it+T_i-st_i, st_i
    [ new_lb_j = \max(new_lb_j, lb_j(i))
    new_ub_j = \min(new_ub_j, ub_j(i))
    /* prologue loop for iterator j */
    for j=it, it+T_j-st_j, st_j
      [ S(i, j)
    /* full-tile loop for iterator j */
    for jt=new_lb_j, new_ub_j-T_j+st_j, T_j
      [ Sunrolled-and-jammed(it, jt) with T_i \times T_j
    /* clean-up loop for iterator j */
    for jt=, new_ub_j, st_j
      [ Sunrolled-and-jammed(it, jt) with T_i \times 1
    /* epilogue loop for iterator j */
    for i=it, it+T_i-st_i, st_i
      [ S(i, j)
  new_lb_j = -\infty
  new_ub_j = \infty
  for i=it, ub_i, st_i
    [ new_lb_j = \max(new_lb_j, lb_j(i))
    new_ub_j = \min(new_ub_j, ub_j(i))
    /* clean-up loop for iterator i */
    for i=it, ub_i, st_i
      [ [ S(i, j)
    /* full-tile loop for iterator j */
    for j=lb_j(i), new_lb_j-st_j, st_j
      [ S(i, j)
    /* prologue loop for iterator j */
    for j=, new_ub_j, st_j
      [ [ S(i, j)
    /* full-tile loop for iterator j */
    for jt=new_lb_j, new_ub_j-T_j+st_j, T_j
      [ Sunrolled-and-jammed(i, jt) with 1 \times T_j
    /* clean-up loop for iterator j */
    for jt=, new_ub_j, st_j
      [ S(i, jt)
    /* epilogue loop for iterator j */
    for j=new_ub_j+st_j, ub_j(i), st_j
      [ S(i, j)
Figure 4.7: Tiled Code.

\[
\begin{align*}
S(i, j) \\
S(i, j+st_j) \\
S(i, j+2*st_j) \\
&\ldots \\
S(i, j+(T_j-1)*st_j) \\
S(i+st_i, j) \\
S(i+st_i, j+st_j) \\
S(i+st_i, j+2*st_j) \\
&\ldots \\
S(i+st_i, j+(T_j-1)*st_j) \\
S(i+2*st_i, j) \\
S(i+2*st_i, j+st_j) \\
S(i+2*st_i, j+2*st_j) \\
&\ldots \\
S(i+2*st_i, j+(T_j-1)*st_j) \\
&\ldots \\
S(i+(T_i-1)*st_i, j) \\
S(i+(T_i-1)*st_i, j+st_j) \\
S(i+(T_i-1)*st_i, j+2*st_j) \\
&\ldots \\
S(i+(T_i-1)*st_i, j+(T_j-1)*st_j)
\end{align*}
\]
CHAPTER 5

EXPERIMENTAL EVALUATION OF CLOOG REGISTER TILING AND TLOG

In this section, we discuss our experimental setup and experiments carried out to assess the effectiveness of the implemented tiling approach in CLOOG. We base our comparison over state-of-the-art tiled code generator, TLOG. As discussed earlier, TLOG is a polyhedral tiled code generator that can generate parametric tiled code for perfectly nested loops.

We use a set of six benchmarks that include linear algebra kernels and stencil computations, as listed in Table 1. As pointed out earlier, due to data dependences, skewing and other unimodular transformations may be needed to make rectangular tiling valid; the need for such skewing transformation is indicated in the table. We made use of a convenient feature of the Pluto system—the option –opt, which causes Pluto to simply transform the code without tiling, but using exactly the same scheduling functions that would have been used to tile the code. This ensures that any skewing needed to enforce rectangular tileability of the code is performed. Intermediate CLooG files generated by Pluto are used as inputs for TLOG and CLOOG, ensuring that all three systems perform tiling on an identically pre-processed version of the input code.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Rectangular Iteration Space</th>
<th>Requires Skewing</th>
</tr>
</thead>
<tbody>
<tr>
<td>DYSRK</td>
<td>Symmetric Rank K Update</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>DTRMM</td>
<td>Triangular Matrix Multiply</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>MATMUL</td>
<td>Matrix Multiplication</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>MVMUL</td>
<td>Matrix Vector Multiply</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>SPR</td>
<td>Rank 1 of Symmetric packed matrix</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 5.1: Experimental Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>DSYRK</th>
<th>DTRMM</th>
<th>MATMUL</th>
<th>MVMUL</th>
<th>SPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLOG</td>
<td>15.2</td>
<td>18.3</td>
<td>30.1</td>
<td>19.6</td>
<td>20.4</td>
</tr>
<tr>
<td>CLOOG (Regtile)</td>
<td>16.3</td>
<td>22.0</td>
<td>28.5</td>
<td>19.3</td>
<td>19.4</td>
</tr>
</tbody>
</table>

Table 5.2: Best Code Execution Times (Time in milliseconds)

The results were taken on a quad-core Intel Core 2 Quad Q6600 CPU clocked at 2.4 GHz (1066 MHz FSB) with a 32 KB L1 D cache, 8MB of L2 cache (4MB shared per core pair), and 2 GB of DDR2-667 RAM, running Linux kernel version 2.6.22 (x86-64). ICC 10.0 is the primary compiler used to compile the source-to-source transformed codes; it was run with -fast; the -fast option turns on -O3, -ipo, -static, -no-prec-div on x86-64 processors these options also enable auto-vectorization in icc.

We try all possible combinations of register tile sizes with values of 2, 4,6 and 8. In Table 5.2, we show the best execution times of register tiled codes. We note here that TLOG does not unroll or register tile its tiled code. We do a post processing on the tiled code generated by TLOG using PRIMETILE, by using annotations to fully unroll the full tiles that are separated by the TLOG tiled code generation algorithm.
<table>
<thead>
<tr>
<th></th>
<th>DSYRK</th>
<th>DTRMM</th>
<th>MATMUL</th>
<th>MVMUL</th>
<th>SPR</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLOG</td>
<td>(8 x 8 x 4)</td>
<td>(8 x 8 x2)</td>
<td>(8 x8 x8)</td>
<td>(8 x 8)</td>
<td>(4 x 4)</td>
</tr>
<tr>
<td>CLOOG (Regtile)</td>
<td>(8 x 8 x 8)</td>
<td>(8 x 8 x 8)</td>
<td>(8 x8 x4)</td>
<td>(8 x 8)</td>
<td>(8 x 1)</td>
</tr>
</tbody>
</table>

Table 5.3: Best Reg-Tile sizes

For rectangular iteration spaces like that of MATMUL, MVMUL and SPR, we find that CLooG (RegTile) has better execution times than TLOG. But DTRMM and DSYRK have better execution times than ClooG-regtiled is because we use Primetile to register tile TLOG generated code - and Primetile uses static calculation of loop bounds for prolog and epilog loops, whereas, in the implementation of register-tile algorithm in CLooG, we use explicit run-time scanning to calculate loop bounds for prolog and epilog loops which traverse the non-rectangular tiles.

CLooG is a powerful state of the art code generator. Tools that we know like Pluto, TLOG, HiTLOG that implement various transformation in the polyhedral framework use CLooG as their code generator. The addition of register tiling algorithm to CLooG can result in improved execution times for the code generated by these tools.
APPENDIX A

APPENDIX A  CLOOG AST DATA STRUCTURES AND FUNCTIONS

In this section, we describe the data structures used by the CLooG AST (CLAST) to represent and process a code generation problem.

A.1 Statement

clast_stmt

struct clast_stmt {
    enum { stmt_ass, stmt_block, stmt_for,
        stmt_guard, stmt_user, stmt_root, stmt_annot } type;
    struct clast_stmt  *next;
};

Every node in the AST represents a statement and is one of the following types:

stmt_root  represents the head of the AST.
stmt_assignment  represents an assignment statement
stmt_block  represents a block

35
stmt_for, - represents a for loop
stmt_guard  if conditional
stmt_user  - user defined statement
stmt_annot  - annotation statement

There is a structure corresponding to each type mentioned above. Every structure has within it a struct clast_stmt. This allows easy type casting within statements. The head of the AST is given by a statement of type stmt_root. Every other statement is one of the other types mentioned above. The next pointer points to the next statement in the program. Every node in the AST, is first checked for its type and then based on the type, it can be type cast to its own struct.

A.1.1 Root

struct clast_root {
    struct clast_stmt    stmt;
    CloogNames *        names;
};

The head of the AST is given by a statement represented by struct clast_root. The stmt member, in this case, has its type set to clast_root. The names pointer points to names of iterators and parameters used in the program. The CloogNames structure is described by the following structure:

struct cloognames
{
    int nb_scalars ;    /**< Scalar dimension number. */
    int nb_scattering ; /**< Scattering iterator number. */

A.1.2 Assignment Statement

An assignment statement is represented by the above struct. The stmt member, in this case, has its type set to *stmt_assignment*. An assignment has two parts: LHS and RHS and are represented by LHS and RHS pointers. LHS pointer points to any variable represented by a string and is used to represent a variable. RHS pointer points to a struct *clast_expr* and is used to represent an affine expression. An affine expression is represented by the structure struct *clast_expr*.
For example, if $a = b + c$ is an assignment statement

\[
\text{stmt.type} = \text{stmt_assignment}
\]

LHS $\rightarrow$ a

RHS $\rightarrow$ b+c

### A.1.3 Block Statement

```c
struct clast_block {
    struct clast_stmt stmt;
    struct clast_stmt* body;
};
```

This structure is used to represent statement block. The body pointer points to a NULL terminated list of statements in a block. The stmt member, in this case, has its type set to `stmt_block`.

### A.1.4 For Statement

```c
struct clast_for{
    struct clast_stmt stmt;
    const char * iterator;
    struct clast_expr * LB;
    struct clast_expr * UB;
    value stride;
    struct clast_stmt * body;
};
```
This structure used to represent a for loop-nest. The stmt member, has its type set clast_for. The iterator is represented by a string pointer. LB pointer points to a lower bound expression, UB points to an upper bound expression. The stride for a for loop is given by an integer. The body pointer points to a block statement. It points to a NULL terminated list of statements in the body. So, a loop nest (a sequence of for loops) is given by subsequent for statements in the loops body.

### A.1.5 User Defined Statement

```c
struct clast_user_stmt {
    struct clast_stmt stmt;
    CloogStatement * statement;
    struct clast_stmt * substitutions;
};
```

This structure is used to define user defined statements like S1, S2 etc. The stmt type in this case will be stmt_user. The substitutions is a NULL terminated list of statements.

For example, consider the user defined statement, $S(i, j)$. The substitutions like $i, j$ can be arbitrarily complex statements like $i = i + 2 \times j$ etc.

The CLooG Statement is represented by the following structure:

```c
struct cloogstatement {
    int number ;/* The statement unique number. */
    void * usr ;/* A pointer for library users convenience. */
    struct cloogstatement * next ; /* Pointer to the next statement with the
A.1.6 Expression

There are three types of expression

1. Term
2. Binary
3. Reduction

struct clast_expr {
    enum { expr_term, expr_bin, expr_red } type;
};

There is a structure corresponding to each type mentioned above. Every structure has within it a struct clast_expr. This allows easy type casting within expressions

A.1.7 Term Expression

clast_term {
    struct clast_expr   expr;
    Value               val;
    const char *        var;
};
A term expression is just a term like a or b. For example, in an expression, \( a + c \), a and c are term expressions.

### A.1.8 Reduction Expression

```c
enum clast_red_type { clast_red_sum, clast_red_min, clast_red_max }; 

struct clast_reduction {
    struct clast_expr expr;
    enum clast_red_type type;
    int n;
    struct clast_expr* elts[1];
};
```

A reduction expression is referred to by the struct `clast_reduction`. The type is set to either sum or min or max. n is the number of smaller expressions each expression is made of and elts is the array of those expressions. For example, reduction expression can be of three types

1) sum - \( a + b \) (n=2, elts[0] -> term a and elts[1]- > term b)
2) min - min(a,b,c) (n=3, elts[0] -> term a and 
   elts[1]- > term b, elts[2] -> term c)
3) max - max(a,b) (n=2 , elts[0] -> term a and elts[1]- > term b)

### A.1.9 Binary Expression

```c
enum clast_bin_type { clast_bin_fdiv, clast_bin_cdiv, 
clast_bin_div, clast_bin_mod };

struct clast_binary {
```
struct clast_expr expr;
enum clast_bin_type type;
struct clast_expr* LHS;
Value RHS;
};

A binary expression is referred to by the struct clast_binary. The type is set to one of the following types. LHS and RHS pointers point to a LHS and RHS expression respectively.

A.2 CLAST Functions Description

A.2.1 CLooG-CLAST Conversion function

struct clast_stmt *cloog_clast_create(CloogProgram *program,
                                     CloogOptions *options);

This function takes in matrix representation of the program as input and returns an AST structure corresponding to the input matrix representation, based on the scattering functions and statement iteration domains represented as matrices. AST representation of an input program is simpler to view, and transformations like unrolling that can be performed syntactically can be done with ease.

A.2.2 Allocation and Initialization Functions

struct clast_<name> new_clast_<name>(<parameters list>)
Each CLAST data structure has an allocation and initialization function, for instance, `struct clast_term new_clast_term(value c, char *v)`. These functions return pointers to an allocated structure with fields set in passed parameters. Using those functions is mandatory to support internal management fields.

For example, `struct clast_term *new_clast_term(Value c, const char *v)`

### A.2.3 Memory Deallocation Functions

```c
void free_clast_<name>(struct clast_<name>);
```

Each CLAST data structure has a deallocation function as shown above, where `Structure` and `structure` have to be replaced by the name of the convenient structure for instance `void free_clast_term(struct clast_term *t)`. These functions free the allocated memory for the structure provided as input. They free memory recursively, i.e. they also free the allocated memory for the internal structures. Using those functions is mandatory to avoid memory leaks on internal management fields.

### A.2.4 CLAST Utility Functions added

#### Copy Functions

We have added functions to do deep copy on any input AST node. The copy functions have the prototype similar to one shown here:

```c
struct clast_stmt *copyStatement(struct clast_stmt *stmt)
```

This function, determines the statement type and traverses the AST from the given node downwards and returns a pointer to another newly allocated AST node, which if syntactically parsed will produce the same code as input node.
Replace Functions

We have added functions, that will replace identifiers (or symbols) on any input AST node. Their prototype is similar to one shown below:

```c
void replaceIdentifier( struct clast_stmt *stmt,
                        char *search_str,
                        char *replace_str)
```

This function determines the statement type and traverses the AST from the given node downwards and returns a pointer to the same node with all variables matching src string replaced with dest.

These functions are extremely useful utility functions to make development of transformations easy, using CLAST.
APPENDIX B

IMPLEMENTATION OF REGISTER TILING IN ROSE

This appendix describes the register tiling algorithm as explained in Chapter 4, using ROSE. Details about SAGE nodes and APIs will be filled in as necessary.

ROSE is an open source compiler framework suited to build custom tools for static analysis, source to source program transformations, performance analysis and cyber-security. ROSE consists of front-ends, a mid end and backends that generate unparsed source code. The intermediate representation used in ROSE is high level to build an Abstract Syntax Tree that is well suited to source to source transformations.

The Sage III IR used by ROSE is an intuitive, object-oriented IR with several levels of interfaces for building source-to-source translators. All information in the original application is carefully preserved in its ROSE AST, including C preprocessor control structure, source comments, source position information, and template information (e.g., template arguments).[5]

The translator reads a C program, builds the AST internally, performs the register tiling operation on the AST (with the information specified in the command line), generates the source code from AST (unparsing) and calls the backend vendor compiler to compile the generated C code.
The translator is written in C++ language in an object-oriented style. When register tiling C applications, this translator only recognizes and optimizes a particular for-loop that corresponds to the DO loop construct in Fortran programs. Within the ROSE source-to-source compiler infrastructure, such a loop is defined to have the following formats:

\[
\begin{align*}
\text{for (i = lb; i <= ub; i+ = positiveStep) or} \\
\text{for (i = ub; i >= lb; i+ = negativeStep)}
\end{align*}
\]

Here \( i \) is an arbitrary integer variable, \( lb \) and \( ub \) are arbitrary integer expressions, and \( positiveStep \) and \( negativeStep \) are positive and negative integer expressions respectively.

The class \( \text{LoopTransformInterface} \) provides the interface for accessing the intermediate representation of an arbitrary compiler, and the pointer reference \( \text{AstNodePtr} \) represents an arbitrary code fragment to be transformed.

To effect changes to the input source code, modifications to the AST are done by rewriting portions of the AST. The AST is the single intermediate form manipulated by the preprocessor and we make changes to the intermediate form by using the \( \text{AstInterface} \). The \( \text{AstInterface} \) is a layer of APIs which presents sophisticated APIs to insert, replace and remove nodes of different types, thus enabling the programmer to focus on algorithm details and not to have to deal with lower level data structures and functions.

The translator uses the AST query library to generate a list of function declarations for any input program. The

\[
\text{NodeQuery::querySubTree(sageProject, V_SgFunctionDeclaration)}
\]
returns a list of function declarations in the program. The result of the AST Node query on the AST is a STL list of IR nodes. We then use $SgFunctionDeclaration$ class and its member functions to get a functions definition. We then iterate through this list of statements and if a statement type is a for statement given by $SgForStatement$ node, we register tile it by calling the $transformLoop()$ function which has the following prototype:

```
bool RegTiling::transformLoop ( AstInterface& fa, const AstNodePtr& s, 
      AstNodePtr& r, 
      list<UnrollList *>outer_unrolls )
```

### B.1 Rectangular Loop

In the $transformLoop()$ function, at any depth, we first decide if the loop iterators at that level is dependent on any outer loop iterators. If it is not dependent on any outer loop iterators, we divide the loop into

1. Main Loop
2. Cleanup Loop

Here we create two new AST nodes using $CreateLoop()$ defined on $AstInterface$. It has the following prototype:

```
AstNodePtr AstInterface::CreateLoop( const AstNodePtr& _cond, 
                                      const AstNodePtr& _body)
```

The lower bound of the main loop and upper bound of cleanup loop are the same as the one in the original loop and are created using the $CopyAstTree()$ functions on
the AstInterface. These functions do a deep copy of the AST under the node. Their prototype is as follows:

\[ \text{AstNodePtr AstInterface :: CopyAstTree( const AstNodePtr &}_\text{orig)} \]

The new upper bound expression for the main loop is calculated as shown in Figures (4.5) and (4.6). The following APIs and classes are used to create different expressions used for lower and upper bounds for loops.

We then recursively call transformLoop() function on main and cleanup loops. We add the unrolled main loop and cleanup loops into a block using CreateBlock() on AstInterface. The following APIs are used to create a block statement:

\[ \text{AstNodePtr AstInterface::CreateBlock( const AstNodePtr& }_\text{orig)} \]
\[ \text{void AstInterface::BlockAppendStmt( AstNodePtr& }_\text{b,} \]
\[ \hspace{1cm} \text{const AstNodePtr& }_\text{s)} \]

A new AST node which is a block statement containing the unrolled loop body is created and it replaces the input AST node using the ReplaceAst() API:

\[ \text{bool AstInterface::ReplaceAst( const AstNodePtr& }_\text{orig,} \]
\[ \hspace{1cm} \text{const AstNodePtr& }_\text{n)} \]

**B.2 Non-Rectangular Loop**

If the loop at that level is dependent on an outer loop iterator, we divide the loop into three parts.

1. Prolog

2. Full Tile

48
3. Epilog

The prolog, full tile and epilog loops are shown in Figures (4.5) and (4.6). The new loop bounds for prolog and epilog loops are calculated using the APIs below. These APIs are all defined by the AstInterface

\begin{verbatim}
AstNodePtr AstInterface::CreateLoop( const AstNodePtr& _cond,
                                   const AstNodePtr& _body)
AstNodePtr AstInterface::CreateAssignment( const AstNodePtr& _lhs,
                                           const AstNodePtr& _rhs)
AstNodePtr AstInterface::CreateBlock( const AstNodePtr& _orig)
void AstInterface::BlockAppendStmt( AstNodePtr& _b,
                                    const AstNodePtr& _s)
AstNodePtr AstInterface::CreateFunctionCall( const AstNodePtr& func,
                                           const AstNodeList& args)
AstNodePtr AstInterface::CopyAstTree( const AstNodePtr &_orig)
\end{verbatim}

We then create a new node which is a block statement. The new block contains all the unrolled loop segments and replaces the original node in the AST.
BIBLIOGRAPHY


