Effects of Loop Tiling using Primetile and Dyntile

Thesis

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

By

Anand Joseph Bernard Selvaraj, B.E.
Graduate Program in Computer Science and Engineering

The Ohio State University

2010

Thesis Committee:

Dr. P. Sadayappan, Advisor
Dr. Atanas Rountev
ABSTRACT

Loop tiling is one of the most important compiler optimization techniques. A multi-level tiled code can improve the running time of the program by maximizing data reuse. PrimeTile and DynTile are tools that are used to generate parametrized multi-level tiled code.

The tile size parameters need not be specified during compilation. This enables runtime optimization of the program. The values assigned to the tile size parameters influence the running time of the program. The choice of the tile size parameters play an important role in the running time of the program.

After generating the multi-level tiled code, it would be ideal to set the tile size parameters to the values that would give the best running time for all the valid permutations of the tile size parameters. In a multi-level tiled code there are several valid permutations for the tile size parameters and hence the search space is huge. An exhaustive search through all the valid permutations defeats the purpose of the search because the time taken for such a search would be much longer than the execution time of the program.

This calls for redefining the problem. After generating the multi-level tiled code, the tile size parameters should be set to the values that give an execution time that is very close to the best running time for all the valid permutations for the tile sizes. To suggest a good search method to find the required tile size parameters, it is necessary to perform an
exhaustive study over the effects on the running time of the program by changing the tile size parameters. This study must be done over different benchmarks.

The tests are done on tiled loops generated by PrimeTile and DynTile. The characteristics of the tiled loops generated by PrimeTile are compared with the tiled loops generated by DynTile. The effects of vectorization on these tiled loops are also studied.
I dedicate my work to my parents Mr. Joseph Bernard Selvaraj and Mrs. Chithra Bernard
ACKNOWLEDGMENTS

I would like to thank my adviser, Dr. P. Sadayappan, for his support and encouragement for my Master’s Thesis. I would also like to thank Dr. Atanas Rountev for serving on my Master’s examination committee.

I would especially like to thank all my friends for the technical inputs that helped me in my Master’s program.

I am grateful to the Department of Computer Science and Engineering for supporting me financially during my Masters program.

Finally I would like to thank my parents for motivating me right from my childhood.
VITA

2008 ......................... B.E, Computer Science and Engineering,
                PSG College of Technology
                Coimbatore, India.

2008-2010 .................. Masters Student,
                Department of Computer Science and Engineering,
                The Ohio State University.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in:

    High Performance Computing       Dr. P. Sadayappan
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>vii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>x</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Parametric Tiling Tools</td>
<td>2</td>
</tr>
<tr>
<td>1.1.1 PrimeTile</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 DynTile</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Vectorization</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Objective</td>
<td>3</td>
</tr>
<tr>
<td>2. Data Aggregation and Analysis</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Data Generation</td>
<td>5</td>
</tr>
<tr>
<td>2.1.1 Benchmarks</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Level of Tiling</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Tile Size Parameters</td>
<td>6</td>
</tr>
<tr>
<td>2.1.4 Compiler</td>
<td>7</td>
</tr>
<tr>
<td>2.1.5 Size of the Data Set</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Consistency Constraints</td>
<td>9</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Percentage of points closer to the best execution time for tiled code generated by PrimeTile</td>
<td>13</td>
</tr>
<tr>
<td>3.2 Percentage of points closer to the best execution time for tiled code generated by DynTile</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Percentage of points closer to the best execution time for tiled code generated by PrimeTile with Vectorization</td>
<td>27</td>
</tr>
<tr>
<td>3.4 Percentage of points closer to the best execution time for tiled code generated by DynTile with Vectorization</td>
<td>29</td>
</tr>
<tr>
<td>4.1 Proximity of the points to the best execution time in the reduced search space with no large outer or inner tile sizes for PrimeTile</td>
<td>33</td>
</tr>
<tr>
<td>4.2 Proximity of the points to the best execution time in the reduced search space with no large outer or inner tile sizes for DynTile</td>
<td>34</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Jacobi Computation Loop</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>FDTD2D Computation Loop</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Lu Computation Loop</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Cumulative percentage of points Vs Ratio with best execution time for PrimeTile</td>
<td>14</td>
</tr>
<tr>
<td>3.2</td>
<td>Jacobi tiled by PrimeTile ordered by ((T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k}))</td>
<td>16</td>
</tr>
<tr>
<td>3.3</td>
<td>Jacobi tiled by PrimeTile ordered by ((T_{1,j}, T_{1,i}, T_{1,k}, T_{2,j}, T_{2,i}, T_{2,k}))</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Jacobi tiled by PrimeTile ordered by ((T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j}))</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>FDTD2D tiled by PrimeTile ordered by ((T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k}))</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>FDTD2D tiled by PrimeTile ordered by ((T_{1,j}, T_{1,i}, T_{1,k}, T_{2,j}, T_{2,i}, T_{2,k}))</td>
<td>20</td>
</tr>
<tr>
<td>3.7</td>
<td>FDTD2D tiled by PrimeTile ordered by ((T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j}))</td>
<td>21</td>
</tr>
<tr>
<td>3.8</td>
<td>LU tiled by PrimeTile ordered by ((T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k}))</td>
<td>22</td>
</tr>
<tr>
<td>3.9</td>
<td>LU tiled by PrimeTile ordered by ((T_{1,j}, T_{1,i}, T_{1,k}, T_{2,j}, T_{2,i}, T_{2,k}))</td>
<td>23</td>
</tr>
<tr>
<td>3.10</td>
<td>LU tiled by PrimeTile ordered by ((T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j}))</td>
<td>23</td>
</tr>
<tr>
<td>3.11</td>
<td>Cumulative percentage of points Vs Ratio with best execution time for DynTile</td>
<td>25</td>
</tr>
</tbody>
</table>
3.12 FDTD2D tiled by DynTile ordered by $(T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})$  26
3.13 LU tiled by DynTile ordered by $(T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})$ . .  26
3.14 Cumulative percentage of points Vs Ratio with best execution time for PrimeTile with vectorization 28
3.15 Cumulative percentage of points Vs Ratio with best execution time for DynTile with vectorization 29
4.1 Cumulative percentage of points Vs Ratio with best execution time for PrimeTile after removing large tile sizes 33
4.2 Cumulative percentage of points Vs Ratio with best execution time for PrimeTile after removing large tile sizes 35
CHAPTER 1

INTRODUCTION

There are a number of transformations that can be done on a program. When a program contains loops that are nested in multiple levels, there are some transformations that can be done to optimize the loops [3]. Some of the popular loop transformations are Loop unrolling, Loop permutation and Loop Tiling. We will concentrate on the tiling or blocking of loops.

Tiling is done to optimize the program for data locality. When a nested loop is tiled, blocks are created which can exploit the data locality and reuse of data [2]. Such tiles or blocks will improve the performance of the nested loops by a significant factor. Tiling can be done in multiple levels and in different sizes and there there are multiple ways to tile the loops. Some tiling scheme might be efficient for a benchmark while another tiling scheme might degrade the performance of the loop. A tiling scheme that optimizes some benchmark really well might not work good for another benchmark.

When the size of the tiles used in the tiled loops is passed as a run time parameter, the tiling is called parametric tiling. The objective of this study is to generalize the optimization of loops with respect to the tile size parameters. Based on the generalization, we expect to learn about selecting tile size parameters that optimize the program at a desired level.
There are multiple ways to transform loops into tiled loops. Let us look at some of the methods.

1.1 Parametric Tiling Tools

In programs which have loops which are nested deep, it is difficult to manually transform the loops into tiles. We have parametric tiling tools that can perform multi-level parametric tiling on input programs. Each of the tiling tools uses a different scheme to transform the loops into tiles. We will use the following two parametric tiling tools for our study.

1.1.1 PrimeTile

PrimeTile was the first parametric tile generator. The tiling is done in multiple stages named prolog, epilog and a loop nest for each level of nesting in the untiled code [2]. The tiled code obtained from PrimeTile is sequential and are not readily parallelizable [2].

1.1.2 DynTile

DynTile is another parametric tiling tool. DynTile uses dynamic scheduling approach to schedule tiles for parallel execution [2]. So, DynTile generates tiles that are more suited for parallel execution.

1.2 Vectorization

A vectorized implementation of a program performs the same operation on multiple pairs of operands at the same time [4]. It is one form of parallelism that can be used to speed up the program. It would be interested to study the effects of vectorization on tiled programs. We are looking at parametrically tiled programs. It will be interesting to know
whether vectorization produces effect on the parametrically tiled program regardless of the values of the tile size parameters. If not, the effect of vectorization will different for different values of the tile size parameters. In that case we have to look for trends that could reason the effects of vectorization on different tile size parameters.

1.3 Objective

The primary objective of this study will be to generalize the performance of the tiled code with respect to the tile size parameters and hence suggest a method to select tile size parameters resulting in good execution times. The secondary objective is to determine whether the values of the tile size parameters influence the effect of vectorization on the tiled code and to generalize the effect of tile size parameters on vectorization.
CHAPTER 2

DATA AGGREGATION AND ANALYSIS

The objective of this analysis is to study the effects of tile size parameters on the running time of a multi-level tiled code. To perform such an analysis, it is necessary to have the running times of the multi-level tiled code for tile size parameters over a given range. The data used for the analysis must be large enough to establish a hypothesis. While generating the data, care needs to be taken to ensure the correctness of the data. All executions of the program must be done in similar conditions. This will make sure that the trends observed from the data are because of the tile size parameters and not because of any other external factor.

After generation, there are multiple ways to view the data. We need to select a view that is appropriate for the analysis. Consider an analysis of the running time based on some parameter. An appropriate view that highlights changes in performance with respect to that parameter makes the analysis much easier than a consolidated view. The conclusions drawn from these analysis will improve the chances of selecting the tile size parameters that give a near optimal performance.
2.1 Data Generation

In this section we will look at the data that is generated for the study. As described earlier, the data needs to be big enough to draw conclusions from it. We are going to generate tiled loops for programs using the two tiling tools PrimeTile and DynTile. This will enable us to compare the performance of the parametrized tiled loops generated by PrimeTile and DynTile. We can use multiple benchmarks and check whether the PrimeTile-DynTile trend holds good for all the benchmarks.

We also vectorize the tiled loops to study the effect of vectorization with respect to tile size parameters. This study will help us decide whether vectorization introduces any complications in selecting the tile sizes or not. If the effect of vectorization is different for different tile sizes, it will be interesting to find out whether vectorization increases or decreases the chances of finding a tile size combination which gives near optimal performance.

Finally we look at the size of the data generated. There are multiple benchmarks which are tiled using primetile and dyntile. We execute those tiled benchmarks with and without vectorization over a number of defined valid permutations. This gives rise to multiple cases. In each case we map the running times of the program with the tile size permutations.

2.1.1 Benchmarks

Three programs are used as benchmarks for the study. They are Jacobi (Fig 2.1), FDTD2D (Fig 2.2) and LU (Fig 2.3). The level of nesting of the loops in all of the benchmarks is equal to three. For the same level of tiling we get same number of tile size parameters for all the benchmarks. This makes the comparison of the trends between the benchmarks straightforward.
\begin{verbatim}
JACOBI COMPUTATION LOOP ()
for(t = 0; t < tsteps; t++)
{
   for(i = 2; i < n - 1; i++)
      for(j = 2; j < n - 1; j++)
         B[i][j] = 0.2 * (A[i][j] + A[i][j - 1] +
   for(i = 2; i < n - 1; i++)
      for(j = 2; j < n-1; j++)
         A[i][j] = B[i][j];
}
\end{verbatim}

Figure 2.1: Jacobi Computation Loop

By studying the performance trend in three benchmarks we can find whether the trend is likely to occur in other benchmarks too. If some trend is found in one benchmark and it doesn’t hold true for another, that trend can be discarded as a benchmark specific trend. We have to note that it is possible to that a trend which holds true for all the three benchmarks might not work for some other benchmark, but it is very likely that it holds true for many other benchmarks.

2.1.2 Level of Tiling

Two dimensional tiled code is generated for each of the benchmarks using PrimeTile and DynTile. So there are two two-level tiled codes generated for each benchmark. The tiled code is executed over a range of tile size permutations.

2.1.3 Tile Size Parameters

Two-level tiled programs have been generated for each of the benchmarks. As mentioned above, the level of nesting in all the benchmarks prior to tiling is equal to three. This
Figure 2.2: FDTD2D Computation Loop

gives three outer tile size parameters and three inner tile size parameters for each program generated by PrimeTile and DynTile. In total there are six tile size parameters for each tiled program.

Each tile size parameter is varied from 2 to 512 by skipping the odd powers of two. So each tile size parameter takes five possible values. In total there are $5^6$ permutations but not all of them are valid. There is a constraint that the outer tile size of a loop must be integrally divisible by the inner tile size of the same loop [1]. This makes the number of valid tile size permutations less than $5^6$. We can look at the exact number of valid tile size permutations that are used to generate the data in the section 2.1.5.

2.1.4 Compiler

As discussed earlier we are going to execute each tiled program with and without vectorization. We use Intel’s icc compiler version 11.1 which has support for vectorization.
LU COMPUTATION LOOP ()

for(k = 0; k < n; k++)
{
    for(j = k + 1; j < n; j++)
        A[k][j] = A[k][j] / A[k][k];
    for(i = k + 1; i < n; i++)
        for(j = k + 1; j < n; j++)
}

Figure 2.3: Lu Computation Loop

We compile the program using ’-fast’ option which is an optimization flag. The option ’-vec’ is used to enable auto vectorization. The option ’-no-vec’ is used to compile the code without vectorization. The option ’-no-vec’ is used to compile the code without vectorization. The tile size parameters are set as a compile time constant and they can be set using the ’-D’ option during compilation. In essence the compiled code for the same tiled program is the same for all the tile size permutations except for the value of the tile size constants.

2.1.5 Size of the Data Set

In the previous sections we have looked at the number of benchmarks, tile generation tools and vectorization options. There are three different benchmarks. Two two-dimensionally tiled loops are generated for each of the benchmarks, one using PrimeTile and the other using DynTile. There are two vectorization options ’-vec’ and ’-no-vec’ which are used during compilation. In total there are 12 different programs that are executed over all the valid tile size permutations specified in section ”‘Tile Size Parameters’”.

8
We create a mapping between the tile size parameters and the Execution times for each of these 12 programs. Each mapping will contain as many entries as the number of tile size permutations that are used during the executions. In section 2.1.3 it is mentioned that each tile size parameter can take any value from 2 to 512 without the even powers of 2 and the inner tile size of a loop must integrally divide the outer tile size of the same loop. There are 15 such valid (inner tile, outer tile) pairs for each loop and there are 3 loops. The number of valid tile size permutations is $15^3$.

To summarize, we have 12 mappings $15^3$ pairs of tile size permutations and execution times each.

### 2.2 Consistency Constraints

The data generated should be accurate enough to ensure that any trends seen during the study is because of the changes in tile sizes and not because of any external factors. To avoid the influence of any external factors, it is necessary to identify such external factors.

There are 12 mappings with $15^3$ entries in each mapping. A total of $12 \times 15^3$ executions is required to generate the required data. The conditions during the tiled-code generation, compilation and execution of the program must be maintained as similar as possible for all the executions.

Let us look at some of the possible external factors during various stages of the generation that might influence the running time of the executions.

#### 2.2.1 Tiled Code Generation

For each benchmark two two-dimensionally tiled codes are generated. One by PrimeTile and the other by DynTile. There are 6 tiled programs generated. Each generated tiled code will be executed $2 \times 15^3$ times using different tile size permutations instead of generating
a different tiled code for each tile size permutation. By reusing the same tiled program for 2 * 15^3 executions we minimize the chances of introducing changes to the program during tiled code generation.

2.2.2 Compilation

Every tiled code is compiled with the same compile time options except for the -vec option and the -D option for the tile size parameters. This ensures that there is no inconsistency introduced because of compiler optimizations. Any change introduced due to -vec is because of the vectorization. Since we are interested to study the effects of vectorization, this is desirable. The only other change in the compiled code would be the compile time constants for the tile size parameters. The compilation is done using the same version (v 11.1) of the Intel icc compiler in the same machine to avoid any other changes that compiler might introduce into the code through compiler optimizations.

2.2.3 Execution of Tiled Code

The execution times might be affected by the other processes that might be executing at the same time. The execution times are comparable only if workload on the processor caused by other processes is constant during all the 12 * 15^3 executions. To ensure this, the processes are allocated to a dedicated node of the glenn cluster of Ohio Supercomputer Center. allocating the processes to a dedicated node avoids the influence of other processes during any of the 2 * 15^3 executions.

Since the order of execution of the 12 * 15^3 programs is not a concern, 12 * 15 jobs were created to execute 15^2 programs each. Each of the 12 * 15 jobs were allocated to a separate
node with one processor per node. The architecture of each processor was identical. Submitting the processes using $12 \times 15$ jobs sped up the data collection without compromising the consistency of the running times.

Now that we have generated the required data, in the next chapter we interpret the data and look for trends in the execution times.
Chapter 3

Interpretation and Analysis

Ch4 We now have the execution times for $15^3$ tile size permutations each for 12 different programs. The 12 different programs are from different benchmarks generated from PrimeTile and Dyntile which are executed with and without vectorization. Each of the 12 programs can be analyzed one at a time to look at the effects of the tiled code generation and vectorization.

For each study the data is viewed in a convenient and appropriate view that would make the study easier.

3.1 Chances of Randomly Picking a good tile size permutation

The loops in a program are tiled with the intention of speeding up the execution times of the program. After the tiled loops are generated the tile size parameters should be chosen. A good tile size permutation must be selected from the given search space. The most naive search would be to randomly pick a point from the search space. If there a lot of tile size permutations that are closer to the best execution time within the search space, then there is a good chance that a randomly selected tile size permutation would give a near optimal execution time. Let us see how easy it is to randomly select a good tile size permutation in the following cases.
<table>
<thead>
<tr>
<th>Closeness to best running time</th>
<th>Jacobi</th>
<th>FDTD2D</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>1.066667%</td>
<td>1.037037%</td>
<td>0.3555556%</td>
</tr>
<tr>
<td>20%</td>
<td>2.488889%</td>
<td>3.496296%</td>
<td>1.007407%</td>
</tr>
<tr>
<td>30%</td>
<td>4.325926%</td>
<td>9.659259%</td>
<td>1.925926%</td>
</tr>
<tr>
<td>40%</td>
<td>9.896297%</td>
<td>17.68889%</td>
<td>3.437037%</td>
</tr>
<tr>
<td>50%</td>
<td>16.94815%</td>
<td>24.17778%</td>
<td>7.318519%</td>
</tr>
<tr>
<td>60%</td>
<td>22.1037%</td>
<td>29.21482%</td>
<td>11.22963%</td>
</tr>
<tr>
<td>70%</td>
<td>28.02963%</td>
<td>34.54815%</td>
<td>13.62963%</td>
</tr>
<tr>
<td>80%</td>
<td>32.2963%</td>
<td>40.35556%</td>
<td>16.32593%</td>
</tr>
<tr>
<td>90%</td>
<td>37.39259%</td>
<td>44.77037%</td>
<td>18.63704%</td>
</tr>
<tr>
<td>100%</td>
<td>42.45926%</td>
<td>47.67407%</td>
<td>20.59259%</td>
</tr>
</tbody>
</table>

Table 3.1: Percentage of points closer to the best execution time for tiled code generated by PrimeTile

### 3.1.1 PrimeTile

Out of the 12 different programs executed over the tile size permutations, 6 programs have the tiled code generated by PrimeTile. We can look at the three programs which are executed without vectorization.

The table 3.1 shows the % of tile size permutations for all benchmarks tiled by PrimeTile that are closer to the best execution time. We can see that the probability of randomly selecting a good point for the LU benchmark is worse than the cases for Jacobi and FDTD2D. The probability of randomly selecting a tile size permutation that will complete the execution at most 10% slower than the best case is around 1% for Jacobi and FDTD2D benchmarks. With a 0.35% chance to pick a similar point, it is almost three times difficult to pick such a point for LU.

If the bar is lowered to find a point that completes the execution at most 50% slower than the best case, we get better results for Jacobi and FDTD2D. With a 16.94% chance,
Figure 3.1: Cumulative percentage of points Vs Ratio with best execution time for PrimeTile

any 6 random tile size permutation for Jacobi is expected to contain a point that gives a running time within the expected limit. FDTD2D has a 24.14% chance of finding such a point. The expected number of random points to be selected to get an execution time within the given range is 4.14. For LU the chance of finding such a point is 7.32%.

From the above analysis we can see that random selection is not very successful for some benchmarks. If the search space is reduced by omitting a lot of bad points, the percentage of good points in the reduced search space will improve. This will improve the chances of finding a good point through random selection. For finding such an reduced search space, we need to find a bunch of bad points that can be removed from the search space.
3.1.1.1 Jacobi by PrimeTile

Let us analyse the results we obtained for the non-vectorized execution of Jacobi benchmark tiled by PrimeTile. The best and worst execution times over all the tile size permutations are 5.25\,s and 38.56\,s respectively. The average of all the execution times over the tile size permutations is 12.60\,s. In this case the choice of the tile size parameters plays a very important role in the execution time of the tiled code. By selecting the appropriate tile size permutation, the program can execute up to 634\% quicker than the execution with a bad choice of tile sizes. The presence of this phenomenon in the other benchmarks will support the role of the tile size parameters in the process of optimization.

We can also see that the average execution time is closer to the best execution time than the worst execution time. An average execution time is around 140\% slower than the best execution time and is 206\% quicker than the worst case. From these values we do not see the distribution of the tile size parameters.

To estimate the probability of selecting a good tile size permutation at random, we need to know the distribution of the points. From the given best case, worst case and average we can assume two possibilities. Either there points are crowded near the average and only some points are scattered along the best and worst cases or vice versa. If more number of execution times are closer to the average execution time, a randomly selected tile size permutation would most give an execution time closer to the average. Even if more number of points are selected, the chances of an execution time closer to the best time is not very high. In the other hand, if the points are crowded near the best and worst execution times, a random selection will most likely give a very good or a very bad execution time. Chances to get an execution time closer to the average execution time are low. When a bunch of random
points are selected, there is a very high chance that it contains a tile size permutation that will give an execution time closer to the best case.

The Figure 3.2 shows the execution times of the Jacobi benchmark tiled by PrimeTile. The tile size permutations \((T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k})\) are lexicographically ordered. \(T_{n,x}\) is the size of the size of the \(n^{th}\) level tile of the \(x\) loop. The \(i\) and \(k\) are the outermost and innermost loops in the untiled benchmark.

![Execution Time Graph](image)

**Figure 3.2:** Jacobi tiled by PrimeTile ordered by \((T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k})\)

From figure 3.2 we can spot a trend at the later part of the X axis. A bunch of points, all of them having an above average execution times, is found. Since the tile sizes are ordered lexicographically, it must be possible to generalize the bunch of points with respect to some tile size parameter. If the X-Axis is divided into 5 equal sections, we can see that the bunch of above average points are present in the last such section. Each of these five sections
corresponds to a particular value for the outer tile size parameter for the *iloop*. The final section corresponds to all the tile size permutations which have the T1,i tile size as 512. From this we can see that all the points with the value of the outer tile size for the outermost loop as 512 result in an execution time slower than the average. We need to look at other benchmarks before we make conclusion based on the largest tile sizes for the outer tile of the outermost loop.

Since the bigger outer tile sizes for the outermost loop has a negative effect on performance, we can look at the effect of other outer tile size parameters.

![Figure 3.3: Jacobi tiled by PrimeTile ordered by (T1,j, T1,i, T1,k, T2,j, T2,i, T2,k)](image)

Figures 3.3 and 3.4 have T1,j and T1,k as the outermost point in the lexicographical ordering. Any patterns seen at the right end of the X-Axis will reflect the influence of the tile sizes T1,j and T1,k when large values are assigned to them. From figures
Figure 3.4: Jacobi tiled by PrimeTile ordered by \((T1_k, T1_i, T1_j, T2_k, T2_i, T2_j)\)

refCh3ImagejikJacobiPrimetileNoVec and refCh3ImagekijJacobiPrimetileNoVec, we see bunches of points which have a slightly greater time than the other points. These bunches are much smaller than the ones seen in figure 3.2. But the pattern observed for \(T1_i\) is observed in \(T1_j\) and \(T1_k\) also but is less pronounced.

Now let us look at the FDTD2D and LU benchmarks.

### 3.1.1.2 FDTD2D by PrimeTile

The best and worst execution times over all the tile size permutations are 7.22s and 54.96s respectively. The average of all the execution times over the tile size permutations is 18.78s. By selecting the appropriate tile size permutation, the program can execute up to 661% quicker than the execution with a bad choice of tile sizes. The average execution time is a little closer to the best execution time than the worst execution time. An average
execution time is around 161% slower than the best execution time and is 192% quicker than the worst case.

The relation between the best, worst and average cases in FDTD2D is similar to the Jacobi case. The only mentionable difference is that the average case is a little farther away from the best case than in the Jacobi case. From table 3.1 we can see that random search in FDTD2D has slight edge over the random search in Jacobi. Let us look at figure 3.5 to find out why.

![Figure 3.5: FDTD2D tiled by PrimeTile ordered by \((T1_i, T1_j, T1_k, T2_i, T2_j, T2_k)\)](image)

In figure 3.5 we can see that the points near the average time are not as crowded as what can be seen in figure 3.2. More points are present below and above the average time which increases the probability of finding points closer to the best and worst cases and reduces the probability of finding times close to the average case. This explains the trend in table 3.1
in which the chances of finding a point which is closer to the best time by within 40% for FDTD2D is almost twice as much as the same case for Jacobi.

At the right end of the X-Axis, we find a bunch of points which are greater than the average time. This is similar to the trend observed in figure 3.2. One notable difference is that, points from the bunch are much greater than the average time than what was seen in figure 3.2 and a few points are present close to the average.

Now we have observed that for both Jacobi and FDTD2D, the outer tile size of the outermost loop when set to 512, does not give results close to the optimal value.

Figure 3.6: FDTD2D tiled by PrimeTile ordered by \((T1_{-j}, T1_{-i}, T1_{-k}, T2_{-j}, T2_{-i}, T2_{-k})\)

In both figures 3.3 and 3.4, we see an expected bunch of points at the right end of the X-Axis.
Figure 3.7: FDTD2D tiled by PrimeTile ordered by $(T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})$

If the same trend is observed in LU, the likelihood of such a trend in any other benchmarks tiled by PrimeTile is high.

### 3.1.1.3 LU by PrimeTile

The best and worst execution times over all the tile size permutations are 0.98s and 31.45s respectively. The average of all the execution times over the tile size permutations is 4.77s. By selecting the appropriate tile size permutation, the program can execute up to 3117% quicker than the execution with a bad choice of tile sizes. An average execution time is around 388% slower than the best execution time and is 559% quicker than the worst case. The best, worst and average cases are wide apart from each other. This increases the difficulty in the search for a near optimal tile permutation. With the average execution time 388% away from the best point the likelihood of points near the best point is very low. This
explains the trend found in table 3.1 where LU has contrasting values compared to Jacobi and FDTD2D.

![Graph showing LU tiled by PrimeTile ordered by (T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k})](image)

Figure 3.8: LU tiled by PrimeTile ordered by (T_{1,i}, T_{1,j}, T_{1,k}, T_{2,i}, T_{2,j}, T_{2,k})

In figure a lot of points are found below the average case but the average is 388% away from the best case. At the right end of the X axis we find a bunch of points similar to the points found in figure 3.2 and figure 3.5. This supports the theory that when T_{1,i} = 512 the tile size permutation is unlikely to give a near optimal execution time. To generalize, when the value of the outer tile size for the outermost loop gets too large, the tiled code is not expected to give a near optimal execution time.

Just like the other benchmarks, both figures 3.9 and 3.10 confirm that when any outer tile variable is assigned with a large value, it negatively influences the running time.
Figure 3.9: LU tiled by PrimeTile ordered by \((T_{1,j}, T_{1,i}, T_{1,k}, T_{2,j}, T_{2,i}, T_{2,k})\)

Figure 3.10: LU tiled by PrimeTile ordered by \((T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})\)
Table 3.2: Percentage of points closer to the best execution time for tiled code generated by DynTile

We have discovered an important pattern regarding the outer tile variables. By avoiding the assignment of larger sizes to outer tile variables, the size of the search space can be reduced. Let us see if the same pattern holds true for dyntile.

### 3.1.2 DynTile

Table 3.2 shows the percentage of points closer to the best performance for the benchmarks generated by DynTile. We can see that for the Jacobi benchmark, the proximity of the points to the best performance is much better than what was found in 3.1.

In the case of FDTD2D, both the PrimeTile and DynTile generations are equally good. The proximity of the points to the best performance for the LU benchmark was bad in the PrimeTile case and is even worser in the DynTile case.

#### 3.1.2.1 Jacobi by DynTile

From table 3.2 we saw that the Jacobi benchmark has about 62% of the points at most 100% slower than the optimal running time. None of the trends with respect to the outer...
tile size variables that were observed in the PrimeTile Jacobi benchmark were seen in the DynTile generation of the tiled Jacobi.

3.1.2.2 FDTD2D by DynTile

No interesting patterns were observed in the FDTD2D benchmark ordering with $T_{1,i}$ and $T_{1,j}$ as the outermost points. However in Figure 3.12 when $T_{1,k}$ is the outermost point, something similar to what was seen in the PrimeTile cases is seen. We can see that the bunch of points found at the right end of the X-axis are located just a little above the other points. The bunch of points are not tightly bound to gather as in the PrimeTile case.
Figure 3.12: FDTD2D tiled by DynTile ordered by $(T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})$

Figure 3.13: LU tiled by DynTile ordered by $(T_{1,k}, T_{1,i}, T_{1,j}, T_{2,k}, T_{2,i}, T_{2,j})$
<table>
<thead>
<tr>
<th>Closeness to best running time</th>
<th>Jacobi</th>
<th>FDTD2D</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>1.155556%</td>
<td>0.7703704%</td>
<td>0.2074074%</td>
</tr>
<tr>
<td>20%</td>
<td>4.740741%</td>
<td>2.340741%</td>
<td>1.214815%</td>
</tr>
<tr>
<td>30%</td>
<td>9.62963%</td>
<td>8.474074%</td>
<td>4.02963%</td>
</tr>
<tr>
<td>40%</td>
<td>16.2963%</td>
<td>16.47407%</td>
<td>6.518518%</td>
</tr>
<tr>
<td>50%</td>
<td>22.66667%</td>
<td>22.63704%</td>
<td>9.511111%</td>
</tr>
<tr>
<td>60%</td>
<td>28.47407%</td>
<td>28.8%</td>
<td>11.97037%</td>
</tr>
<tr>
<td>70%</td>
<td>32.8%</td>
<td>35.28889%</td>
<td>14.04444%</td>
</tr>
<tr>
<td>80%</td>
<td>37.8963%</td>
<td>41.36296%</td>
<td>15.2%</td>
</tr>
<tr>
<td>90%</td>
<td>42.63704%</td>
<td>45.27407%</td>
<td>16.53333%</td>
</tr>
<tr>
<td>100%</td>
<td>47.52592%</td>
<td>47.82222%</td>
<td>17.83704%</td>
</tr>
</tbody>
</table>

Table 3.3: Percentage of points closer to the best execution time for tiled code generated by PrimeTile with Vectorization

### 3.1.2.3 LU by DynTile

A bunch of points with worsner running time than the average are found when the value of $T_{1,k}$ becomes too large. This can be seen in figure 3.13. This behavior is exhibited only for $T_{1,k}$.

### 3.1.3 Vectorization

#### 3.1.3.1 Vectorization on PrimeTile

Table 3.3 shows the effects of vectorization on tiled code generated by PrimeTile. Comparing with the values in refCh3TablePrimeTileNoVec we can decide whether vectorization increases the likelihood of tile size permutations closer to the best case execution time. In the Jacobi benchmark vectorization improves the number of points which are closer to the optimal. This improvement is more pronounced in the points closer to the optimal by 10% and 50%. This is a good sign because it improves the likelihood of selecting points closer to the optimal.
The same trend is found for the FDTD benchmark. But for the LU benchmark vectorization reduces the chances of finding a good point. We can conclude that vectorization affects the execution time of a tiled code and the change in the execution time is dependent on the tile size permutation. But from the above analysis it is not possible to generalize the effect of vectorization on tiled code generated by PrimeTile.

### 3.1.3.2 Vectorization on DynTile

From tables 3.4 and 3.2 it is found that there is not much change for the Jacobi and FDTD2D benchmarks. But for the LU benchmark, vectorization negatively influences the chances of finding a good tile size permutation. Just like the PrimeTile case, we are not able...
Closeness to best running time | Jacobi | FDTD2D | LU |
--- | --- | --- | --- |
10% | 1.777778% | 0.9777778% | 0.3259259% |
20% | 15.17037% | 4.148148% | 0.6814815% |
30% | 27.02222% | 11.40741% | 1.392593% |
40% | 29.06667% | 16.68148% | 2.074074% |
50% | 33.74815% | 27.82222% | 2.281482% |
60% | 39.43704% | 32.44444% | 3.2% |
70% | 45.68889% | 35.91111% | 4.296296% |
80% | 50.04445% | 40.2963% | 5.066667% |
90% | 52.97778% | 42.96296% | 6.074074% |
100% | 55.58519% | 45.71852% | 7.140741% |

Table 3.4: Percentage of points closer to the best execution time for tiled code generated by DynTile with Vectorization

Figure 3.15: Cumulative percentage of points Vs Ratio with best execution time for DynTile with vectorization
to generalize the effect of vectorization on the tiled code with respect to the tile size parameters. But we have collected enough evidence to show that the influence of vectorization on the running time of a tiled code is different for different tile size permutations.

### 3.2 Removing Bad Tile Size Permutations

In Section 3.1 we were able to generalize some tile size permutations where the running time is expected to be much worse than the best case. In this section we suggest a reduced search space by removing permutations which are identified to give longer execution times.

#### 3.2.1 PrimeTile

In section 3.1.1 we observed a pattern with respect to the outer tile size parameter of each loop. When the outer tile size parameter is assigned a bigger value, the running time of the program increases considerably. This provides us a chance to generalize a bunch of tile size permutations which are not optimal. Removing these points from the search space should definitely improve fraction of good tile size permutations. The same trend was observed in all the three benchmarks. This suggests that the chances of the same trend being exhibited in other programs are high.

#### 3.2.2 DynTile

The pattern observed in programs tiled by PrimeTile was very well defined. From section 3.1.2 we observe a pattern that is similar to the one observed in PrimeTile, but is not as strong as the PrimeTile case. Two out of the three benchmarks have a bunch of permutations which have execution times clearly away from the best case when the outer tile parameter for a particular loop is large. In the PrimeTile cases, this behavior was exhibited for the outer tile parameter for all the loops. In the DynTile cases we find
the pattern for the outer tile parameter of only one of the loop. Since the trend is based on only one of the outer tile parameters, it is not straightforward to generalize a method to remove bad permutations. One of the benchmarks didn’t have the bunch of points with high execution times for any outer tile parameter. This suggests that each benchmark might exhibit such a trend with respect to the outer tile parameter of a different loop.

In the next chapter we remove a set of permutations from the search space and estimate the fraction of good permutations from the reduced search space.
CHAPTER 4

REMOVING BAD PERMUTATIONS

In the previous Chapter we looked at the chances of selecting a good tile size permutation by randomly picking a tile size permutation from the defined search space. The chances of selecting a good tile size permutation will improve if we remove some tile size permutations with execution times closer to the worst case from the search space. To remove such bad tile size permutation we need to generalize the bad execution times based on the tile size permutation. In section 3.2 we discussed some trends that were found in the execution times that enabled us to generalize some tile size permutations which give bad execution times. In this chapter, we remove those tile size permutations to create a better search space.

4.1 Removing Bad Permutations for PrimeTile Generated BenchMarks

In section 3.2.1 it was discussed that the tile size permutations with large values for the outer tile parameters of any loop negatively influences the execution times. The first thing that can be done to reduce the search space is to remove permutations that have a large value for any outer tile size parameter. For the given benchmarks we found when outer tile parameters were as large as 512, the execution times were bad. We can create a reduced search space by removing all permutations which have any outer tile size parameter as 512.
<table>
<thead>
<tr>
<th>Closeness to best running time</th>
<th>Jacobi</th>
<th>FDTD2D</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>1.8%</td>
<td>3.5%</td>
<td>1.2%</td>
</tr>
<tr>
<td>20%</td>
<td>5.2%</td>
<td>8.2%</td>
<td>3.2%</td>
</tr>
<tr>
<td>30%</td>
<td>9%</td>
<td>20.5%</td>
<td>5%</td>
</tr>
<tr>
<td>40%</td>
<td>18.1%</td>
<td>32.6%</td>
<td>8.5%</td>
</tr>
<tr>
<td>50%</td>
<td>27.7%</td>
<td>42.9%</td>
<td>16.3%</td>
</tr>
<tr>
<td>60%</td>
<td>34.3%</td>
<td>50.1%</td>
<td>21.5%</td>
</tr>
<tr>
<td>70%</td>
<td>40.3%</td>
<td>59.3%</td>
<td>26.3%</td>
</tr>
<tr>
<td>80%</td>
<td>46.5%</td>
<td>66.8%</td>
<td>30.7%</td>
</tr>
<tr>
<td>90%</td>
<td>52.2%</td>
<td>70.8%</td>
<td>33.1%</td>
</tr>
<tr>
<td>100%</td>
<td>57.6%</td>
<td>72.8%</td>
<td>35.8%</td>
</tr>
</tbody>
</table>

Table 4.1: Proximity of the points to the best execution time in the reduced search space with no large outer or inner tile sizes for PrimeTile

Figure 4.1: Cumulative percentage of points Vs Ratio with best execution time for PrimeTile after removing large tile sizes
<table>
<thead>
<tr>
<th>Closeness to best running time</th>
<th>Jacobi</th>
<th>FDTD2D</th>
<th>LU</th>
</tr>
</thead>
<tbody>
<tr>
<td>10%</td>
<td>21.8%</td>
<td>4%</td>
<td>0.6%</td>
</tr>
<tr>
<td>20%</td>
<td>33.9%</td>
<td>11.6%</td>
<td>2.2%</td>
</tr>
<tr>
<td>30%</td>
<td>41.3%</td>
<td>26.2%</td>
<td>4.3%</td>
</tr>
<tr>
<td>40%</td>
<td>47.3%</td>
<td>38.2%</td>
<td>7.2%</td>
</tr>
<tr>
<td>50%</td>
<td>52.4%</td>
<td>46.1%</td>
<td>9.8%</td>
</tr>
<tr>
<td>60%</td>
<td>57.7%</td>
<td>55.7%</td>
<td>11.9%</td>
</tr>
<tr>
<td>70%</td>
<td>62.9%</td>
<td>64.8%</td>
<td>14.1%</td>
</tr>
<tr>
<td>80%</td>
<td>67.8%</td>
<td>71.5%</td>
<td>16.5%</td>
</tr>
<tr>
<td>90%</td>
<td>70.7%</td>
<td>76.9%</td>
<td>17.8%</td>
</tr>
<tr>
<td>100%</td>
<td>74%</td>
<td>80.3%</td>
<td>20.5%</td>
</tr>
</tbody>
</table>

Table 4.2: Proximity of the points to the best execution time in the reduced search space with no large outer or inner tile sizes for DynTile

Table 4.1 shows the increased fraction of good tile size permutations.

### 4.2 Removing Bad Permutations for DynTile Generated BenchMarks

In section 3.2.2 we discussed that the bad execution times were related to at most one outer tile parameter. Since different benchmarks can be dependent on the outer tile sizes of different loops, it is difficult to generalize the trend. We can reduce the search space by removing permutations that have a large value for any outer tile size parameter. By this we ensure that any bad permutations caused by the large size of any outer tile parameter will be discarded from the search space. Since not all outer tile parameters influence the execution time, some good tile permutations will also be discarded. We are not able to avoid this because, for a new benchmark we will not be able to identify the loop for which the large outer tile causes bad execution times. Similar to the PrimeTile case, create a reduced search space by removing all permutations which have any outer tile size parameter as 512.

Table 3.2 shows the fraction of good tile size permutations has increased.
We see some improvements in the chances from what we saw in table 3.2. We can see that there is a considerable improvement in the probability of selecting a good tile size permutation for all the benchmarks.

4.3 Conclusion

By studying the results obtained from this experiment, we can see that the execution time of a tiled code is negatively influenced by large tile sizes. This behavior was clearly exhibited by the benchmarks tiled by PrimeTile. We can conclude that tile size permutations which have a large value for any tile size parameter are highly likely to result in a longer execution time. In the case of these benchmarks the value of the tile size which results in longer execution times is 512.
In case of programs tiled by DynTile, we were able to find trends similar to what was observed in the PrimeTile cases. But the trends were not as strong as seen in PrimeTile. This is because some of the tile size permutations which had a large tile size resulted in good execution times. Since we were not able to find any other trend that was able to select a bunch of permutations which give bad executions, we proceeded to use the same method to reduce the search space. By discarding all the tile size permutations with large tile sizes, a number of bad points were removed.

The effects of vectorization on tiled code were studied. For all the three benchmarks, the speed up achieved through vectorization was dependent on the tile size permutations.

After the study performed results obtained, we decide that large tile sizes negatively influence performance.
BIBLIOGRAPHY


