Sampling of Dynamic Dependence Graphs for Data Locality Analysis

Thesis

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

By

Gaganjit Singh Jhally

Graduate Program in Computer Science and Engineering

The Ohio State University

2016

Master’s Examination Committee:

Dr. P. Sadayappan, Advisor

Dr. Nasko Rountev
Abstract

Data locality is a critical factor which affects the execution time of applications today. With major advances being made in reducing the computation time of processors, data movement costs have increasingly become a bottleneck in runtime and energy efficiency of current applications. Existing dynamic analysis tools cannot provide any guidance on whether improvement in data movement costs maybe feasible. Lower and upper bounds on data movement costs can offer a solution to this problem. We put forth an approach to addressing this problem by using lower and upper bounds analysis of arbitrary programs. In this thesis, we will be focusing on the upper bound analysis using an existing framework.

The framework has been used to develop tools to assess the data movement costs and help identify changes in execution schedule to increase the performance of an application. The framework achieves this by generating and analyzing a computational directed acyclic graph (CDAGs) for an execution. The size of these graphs can go up to billions of nodes. This thesis highlights how a sampling technique applied to the CDAG can reduce the time taken to analyze a real world application.

The second part of the thesis is motivated in part by the existing difficulty in automating the time tiling of nested loops. Current strategy involves skewing the time loop to improve cache hits. Our strategy was developed to make use of fine grained parallelism offered by OpenMP tasks, and to subsequently utilize it to execute
multiple tiles across time loop iterations, and utilize the speedup due to inherent data locality across time tiles.
Acknowledgments

My deepest gratitude is to Dr. P. Sadayappan and Dr. Nasko Rountev for their keen guidance throughout my research. I am grateful to them for giving me this important opportunity to be a part of their research group. I learnt a lot from them, not only through their courses but from their energy, commitment, and work ethics, which are big inspirations for me and I will always cherish the time spent in their guidance.

Special thanks to Venmugil and Aravind - the two most helpful and smartest people I have ever met. They were extremely patient during our discussions and I have learnt a number of things from them, not only about this domain but also Linux good practices. Venmugil was integral in getting me up to speed on my research topics and helped clear the smallest questions I had with the tool. Aravind has been an awesome mentor, helping me address the strangest questions and being my design discussion peer. I would like to thank my friends and fellow researchers in lab - Shridhar, Kiran, Nisa, Ankut, Wenlei, Prashant, Changwan and Jinsung.

At this point I would like to thank my Dreese lab cohort Meraj, Albert, Kaushik, Shridhar and Ganga for a number of things. From procrastinating chai sessions to long lunches to late night frisbee games and to the occasional midnight oil burning.

Finally, I wouldn’t be what I am today without my parents and brother and my friends who are my greatest support and source of inspiration.
Vita

2009  .................................................. B.E. Electrical Engineering,
       Nanyang Technological University, Singapore
2009-2014  ............................... Field Applications Engineer,
              Accellion, Singapore
2014-2015  ............................... Graduate Research Associate,
              Computer Systems, ACCAD,
              The Ohio State University
2015-present  ......................... Graduate Research Associate,
               Dept of CSE
               The Ohio State University

Fields of Study

Major Field: Computer Science and Engineering
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>iv</td>
</tr>
<tr>
<td>Vita</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>viii</td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Dynamic analysis</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Problem</td>
<td>2</td>
</tr>
<tr>
<td>1.4 Improvements contributed by this work</td>
<td>3</td>
</tr>
<tr>
<td>2. Background</td>
<td>4</td>
</tr>
<tr>
<td>2.1 Dynamic analysis framework</td>
<td>4</td>
</tr>
<tr>
<td>2.1.1 Instrumenting the program</td>
<td>5</td>
</tr>
<tr>
<td>2.1.2 Generating the trace</td>
<td>6</td>
</tr>
<tr>
<td>2.1.3 Constructing the DDG</td>
<td>7</td>
</tr>
<tr>
<td>2.2 Computational Directed Acyclic Graph</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Convex Partitioning Heuristics</td>
<td>9</td>
</tr>
<tr>
<td>2.3.1 Reuse distance analysis</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Convex component</td>
<td>10</td>
</tr>
<tr>
<td>2.3.3 Upper bounds algorithm</td>
<td>10</td>
</tr>
<tr>
<td>3. Sampling based Upper Bounds</td>
<td>12</td>
</tr>
<tr>
<td>3.1 CDAG sampling</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1 Instrumentation</td>
<td>14</td>
</tr>
</tbody>
</table>
3.1.2 Trace analysis ................................................. 14
3.1.3 Graph construction and heuristics ......................... 17
3.2 Evaluation and results ........................................ 18
  3.2.1 Regular Benchmarks ..................................... 18
  3.2.2 SPEC Benchmarks ....................................... 19

4. Time Loop tiling ............................................... 23
  4.1 Introduction .................................................. 23
  4.2 OpenMP tasks ............................................... 23
  4.3 Strategy and implementation ............................... 24
    4.3.1 Jacobi-2D .............................................. 25

5. Conclusion ....................................................... 31

6. Future Work ..................................................... 33

Appendices

A. Running the DDG tool for the sampled space ................ 35

Bibliography .......................................................... 36
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Single-sweep two-point Gauss-Seidel code.</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Jacobi-2d sampled vs non sampled upper bound analysis.</td>
<td>19</td>
</tr>
<tr>
<td>3.2</td>
<td>Jacobi-3d sampled vs non sampled upper bound analysis.</td>
<td>20</td>
</tr>
<tr>
<td>3.3</td>
<td>GemsFDTD sampled upper bound analysis.</td>
<td>21</td>
</tr>
<tr>
<td>3.4</td>
<td>Bwaves sampled upper bound analysis.</td>
<td>22</td>
</tr>
<tr>
<td>4.1</td>
<td>Data dependence element comparison across tiles.</td>
<td>25</td>
</tr>
<tr>
<td>4.2</td>
<td>Comparison of sequential, OpenMP tasks, OpenMP for implementation of Jacobi-2d.</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Comparison of sequential, OpenMP tasks implementation of GemsFDTD.</td>
<td>30</td>
</tr>
</tbody>
</table>
Chapter 1: Introduction

1.1 Motivation

Advances in technology over the last few decades have yielded significantly different rates of improvement in computational throughput compared to the speed of memory access. This mismatch between the maximum operations achieved by a processor and peak data movement rates makes data movement an increasingly critical factor in determining the performance of an application. This divide between computation speeds and data access overheads will only increase in future. Therefore, optimizing the memory access in an application will be an area of focus for application developers. However, existing tools and methods cannot provide guidance to developers about the ways to reduce the data movement overheads.

Hardware counters and static performance analysis tools such as HPCToolkit [5] can help application developers characterize the run-time execution of an application and identify the hot loop regions with a performance bottleneck. However, the above mentioned tools do not provide a way for developers to decouple the causes of such bottlenecks, which could range from sub-optimal code implementation to the large data movement cost associated with the algorithm used in the application itself. This is partly because these tools measure properties of a single execution of the program.
1.2 Dynamic analysis

One of the approaches to addressing the problem mentioned above is to perform dynamic analysis on a given execution and evaluate the properties of the application over a few runs. Once the analysis is done, we can identify the data dependencies of various operations being executed in the program and build a computation directed acyclic graph (CDAG). Each vertex in this graph corresponds to a computation in the application. The CDAG expresses the minimal ordering constraints on computations for correct execution of the algorithm.

In this thesis, we continue the work on an existing dynamic analysis framework [3] that has been used to develop different dynamic analysis tools [4]. One of the upper bound heuristics developed in this framework comes up with a different schedule from the one of a normal execution of the application, having a lower data movement cost. We will be focusing on this specific tool and using it to find and measure the performance of a few of the SPEC benchmarks.

1.3 Problem

The CDAG generated for real-world applications (simulated via SPEC benchmarks) are typically very large graphs. On the basis of their representation, they are represented could easily take disk space in GBs. Reducing the input data set from ref to train in these benchmarks might not be acceptable in other applications. Therefore, there is a need to be able to analyze and run heuristics on these large CDAGs in spite of having limitations on disk space and memory being used by the machine running the analysis. We implemented a sampling algorithm which could
be used to improve the run-time of the analysis and still provide insight into possible improvements using the heuristics already in place.

1.4 Improvements contributed by this work

The overall dynamic analysis framework was developed by [3]. The tools and heuristics which are a part of this framework that help in measuring the data locality were developed by [4] [8] [9]. The work done in this thesis extends this framework by allowing an option to run the heuristics on a sampled CDAG instead of the complete graph. Additionally we ran our sampling implementation on SPEC benchmarks and a few well known algorithms and results are highlighted in Chapter 4. Chapter 2 discusses the existing framework and gives a background of the various tools and heuristics in place. Chapter 3 discusses the implementation details.

The work in this thesis was partly motivated by the bottlenecks we observed while running the heuristics on GemsFDTD and Gromacs SPEC benchmarks. After examining the code for the above mentioned benchmarks we see a time loop iterating over stencil computations. The inner loops can be tiled easily but the time loop prevents gaining any spatial data locality. In this thesis, we decided to explore time tiling strategies using the OpenMP tasking constructs.

The OpenMP tasks allow a user more fine grained control over the dependencies in various parts of an application. This can be leveraged to allow the running of tiles across time loop iterations without having completed all the work in the prior iteration of the time loop. We outline this strategy and its manual implementation in Jacobi-2D and GemsFDTD SPEC benchmark in Chapter 4.

In Chapter 5 we summarize the thesis.
Chapter 2: Background

2.1 Dynamic analysis framework

The first step in the approach is to generate the implementation independent data dependency graph (DDG) from the input program using the existing dynamic analysis library. The existing framework consists of 3 parts: Instrumentation, trace generation, and running heuristics on the trace for analysis.

1. Instrumentation: Instrumenting the input program with function calls that allow us to capture interesting events. The loops to be instrumented can be identified by running a static analysis tool like HPCToolkit on the program.

2. Trace generation: Executing the instrumented program to generate the trace files. This is done with the help of a runtime library that is linked and executed with the original program.

3. Analysis: Analyzing the trace file to generate the DDG.

Since we are dealing with scientific/engineering domain specific applications, we only register floating point operations in the DDG.
The tool uses the LLVM compiler infrastructure [2] to automate the different phases mentioned above. LLVM lets us compile the LLVM intermediate representation (LLVM-IR) of the input program. It supports both C and Fortran languages and has a well defined code representation in the form of LLVM-IR. Additionally, LLVM was chosen because of number of useful features it provides like loop analysis and block analysis.

To give an overview of the above process, the program is first compiled to obtain an LLVM-IR file. This file is instrumented, linked, and executed with the run-time library. The resulting executable produces the trace file containing the record of interesting program events.

2.1.1 Instrumenting the program

The instrumentation phases job is to capture interesting events in the execution of the program. These program events are represented as functions. The run-time library handles these functions(events). When the compiled LLVM-IR file is executed, the markers put in place by the instrumentation will be activated and relay the information which they collect to the run-time library. For example, the instrumented loop would be considered as a function, and the loop entry and loop exit would be marked as program states, and collected by the run-time library.

Each of these events have certain parameters depending on the type of events. Various types of parameters are either user defined or they are already defined in the LLVM infrastructure like basic block, functions etc. The program entities are assigned a unique ID. The run-time library allows the user to instrument the hot regions of an application using the "ddg_start_trace" and "ddg_stop_trace" functions.
Additionally, LLVM also gives us the option to add supplemental information onto the instructions. This data is stored in the IR file and can be accessed again. This is an important feature and lets us read the file while running the heuristics to fetch the additional information.

The events which are captured during the instrumentation phase in the source program are listed below

1. **Load**: This marks the occurrence of load instruction. Parameters: *Instruction ID, Address read by load*

2. **Store**: This marks the occurrence of a store instruction. Parameters: *Instruction ID, Address written by the store*

3. **Basic Block Enter**: This event marks an entry into a basic block. Parameters: *Basic block ID*

4. **Function events**: The events included under this umbrella of functions are function call, function return and function enter. Parameters: *Function address, Instruction ID of the call statement*

5. **Loop events**: These events mark the beginning and ending of a loop. Additionally, they also mark the start and end of loop iterations. Parameters: *Loop ID*

### 2.1.2 Generating the trace

The instrumentation phase is used to insert calls to certain functions in the program. The actual generation of the trace file occurs during the program execution. Binary format is used to store the trace file. This is preferred as it is more space
efficient. Additionally, the file is also compressed using gzip to keep the size of the file small. The trace file generated contains a list of events occurring during the program execution. Each of these events has a unique number for identification. Additionally, the parameter list is also appended to the event. While reading the file, the trace file is uncompressed and read.

2.1.3 Constructing the DDG

A dynamic dependency graph is an abstract representation of an execution of a program. DDG is constructed by utilizing the trace analysis components API’s. When the analysis is taking place, every instruction is represented by a node which also additionally stores all the information like ID, address, and type of instruction. Furthermore, we also store the predecessors of an instruction; that is all the instructions on whose execution the current instruction is dependent. The predecessor information for a node is obtained by traversing through the dependencies using registers. The library stores a last writer for every virtual register. Last writer to a register will be the predecessor of any instruction reading or writing to that register.

Only flow dependencies are represented in the graph. Anti-dependency and output dependency are not considered as these can be removed by transformations of the program. During the construction of the DDG, the whole graph is not stored in memory at any given time, mainly due to the large size of the graph. The client only sees a part of the graph being constructed during the trace playback.

The following steps are used while creating the DDG
1. Predecessor information for a given node is tracked by maintaining a table of
last writers for each virtual register/memory address. This table is used to
lookup the last writers of the operands of an instruction.

The new node is then inserted into the last writer table for the register it is
writing to. If the register had a previous writer, that is destroyed during this
process. The size of this table is to the order of number of instructions.

2. A graph node is created for each instruction. This node stores the ID and
address and list of predecessors.

Lazygraph API's are used while constructing the DDG. VisitNode is used in the
library to add a graph node. The function takes the nodes information, predecessors
and operands of this instruction as arguments.

2.2 Computational Directed Acyclic Graph

Valid scheduling orders for the operations of an algorithm can be modeled us-
ing a schedule invariant abstraction referred to as the computational directed acyclic
graph (CDAG). The CDAG consists of computations as vertices and edges from pro-
ducer instances to consumer instances. Figure 2.1(c) shows the CDAG for the codes
in figure 2.1(a) and figure 2.1(b) for N=6. Even though the relative order of compu-
tations are different between the tiled and the untiled version, the CDAG for both
looks identical. This is due to the computation nodes and identical sets of producer-
consumer relations in both versions of the code.

Definition 1 (CDAG-HK [6]) A computational directed acyclic graph is a 4-tuple
C = (I, V, E, O) of finite sets such that: (1) I \ V is the input set and all its vertices
have no incoming edges; (2) \( E V V \) is the set of edges; (3) \( G = (V, E) \) is a directed acyclic graph; (4) \( V I \) is called the operation set and all its vertices have one or more incoming edges; (5) \( O V \) is called the output set.

DDG provides sufficient information for a CDAG to be constructed. The main difference between DDG and CDAG is that DDG also contain load and store information as vertices, while CDAG only stores computations as nodes. Therefore, we can see that a CDAG construction also requires the tracking of transitive dependencies between various instances in the trace.

### 2.3 Convex Partitioning Heuristics

The algorithms to measure the data locality potential of a given program using dynamic analysis were developed by Fauzia et al[3]. We use the convex partitioning algorithm, which was re-implemented as a tool in this framework, to calculate the
new schedule with lower data movement costs compared to the original execution. Before we dive into the algorithm, we will first define the key parts of the algorithm

2.3.1 Reuse distance analysis

The algorithm uses reuse distance analysis (RDA) as a metric to model data locality[3]. The reuse distance metric of a reference in a trace is defined as the number of distinct memory address accesses between two successive references to the same location.

2.3.2 Convex component

Definition 2 (Convex component) Given a CDAG G, a convex component \( V_i \) in G is defined as a subset of the vertices of G such that, for any pair of vertices u and v in \( V_i \), if there are paths between u and v in G, then every vertex on every path between u and v also belongs to \( V_i \).

A convex partition of a graph G is obtained by assigning each vertex of the graph to a convex component. Furthermore, each vertex in CDAG is assigned to a single convex component, which results in an acyclic convex components graph. These convex components can be executed in any topologically sorted order.

2.3.3 Upper bounds algorithm

The RDA gives data locality costs for a specific execution order. The algorithm tries to find other orders of execution which can give better RDA profile than the original execution. The algorithm works as follows

1. Generate the sequential execution trace for a program
2. Measure the data locality of the sequential trace using RDA
3. Construct a CDAG from the execution trace

4. Identify the convex components of the CDAG which are used to change the schedule of the operations. Convex partitioning is analogous to tiling, insofar running these Convex components grouped up as partitions will provide a better RDA cost.

5. Measure the data locality of the new schedule using RDA

While applying convex partitioning algorithm to the CDAG, we should ensure that if a vertex is being added to the convex component, then all the vertices on the path from the new vertex to vertices inside the component are also added to the component.
Chapter 3: Sampling based Upper Bounds

In Chapter 2, we covered the existing tool and how it was used to measure the upper bounds (UB) of data locality potential of a program.

The approach described in Chapter 2 required the full CDAG to be held in the memory or on disk for the analysis. As we have seen, some of the real-world applications could possibly generate CDAGs containing millions or even billions of nodes. Consequently, the run-time of some of the heuristics could take up to a few hours to complete. Furthermore, the trace file which is generated during the execution of the program (instrumentation), grows into GBs when recording events captured from the application.

Additionally, we have to tune the disk-cache[9] based on the application for which the UB heuristics are being run. This requires the application developer to have an understanding of how the cache mechanism is optimized.

However, a number of scientific/engineering domain specific applications have fairly consistent run-time dependencies, that is the run-time data dependencies do not show variance between iterations. We leverage upon this fact to offer a sampling based approach to measure the UB of an application. By capturing a smaller sample of iteration space, we can draw some useful insights about the full execution of the application.
The following section describes the methodology used while sampling. A section of execution details are captured, depending on how much memory is available for the analysis, and a sub-CDAG is generated based on these details.

3.1 CDAG sampling

We are interested in extracting the CDAG which corresponds to a certain iteration space, without having to record events for the full execution. Specifically we will be capturing the execution details for the first few iterations of each loop in the loop nest being analyzed. The current implementation starts sampling from the beginning of each loop in the loop nest till the sample size specified during the instrumentation phase. This is one of the ways to improve the sampling approach, where the sampling start and stop can be marked anywhere in the area of interest. This is discussed further in Chapter 6.

Listing 3.1: 2d loops

```c
for( i=1; i<100; i++) {
    for( j=1; j<100; j++) {
        S1; //Computation
    }
}
```

In listing 3.1, we are interested in getting a smaller section of the iteration space instead of the complete range which is \( (i,j)-1 \leq i,j < 100 \). We are looking to create a sub-CDAG for the first 10 iterations of each loop, that is \( (i,j)-1 \leq i,j \leq 10 \). We need to be careful at this point to ensure that no edges which are going out or coming into the sampled CDAG section are lost during the construction of the CDAG. The edges need to be preserved in order to ensure that the upper bound being calculated
is accurate. The construction of CDAG is completed in three phases, and we will note the changes made to each part in the following subsections.

### 3.1.1 Instrumentation

Loop trip counters are added to loop headers to capture the iteration count during the execution. Boolean variables are used to keep track of whether the computations are within the sample space or outside of it. Conditional variables are added to each of the loop header in the loop nest to identify iterations falling inside the sample space; if true, then the boolean variable for the computation is set to true. If the loop body has function calls, then the boolean variable will have to be forwarded as well.

### 3.1.2 Trace analysis

During the execution phase, it is not feasible to ignore the computations outside the sample space as this would mean some of the dependencies are not captured during the trace generation. Subsection 2.1.3 highlights how the trace is generated and the graph is constructed from the trace. We cannot ignore any writes to memory outside the sample space, as this would mean the information in last-writer will be outdated. We use a flag to mark any address written by a node from outside the sample space as invalid.

During the analysis phase, when an invalid address is read from the trace file, the entry in last-writer table is marked as invalid. Any computation with operands consisting of invalid addresses will be marked as invalid. As an optimization, we invalidate the address once, until it gets overwritten by a valid address inside the sample space. It is sufficient to invalidate the address only once, within a continuous range of execution outside the sample space.
During this phase, \texttt{numPreds} for each node is set. This includes only the predecessors inside the sampling space. To mark those nodes in the sample space which might use memory addresses written outside of the sample space, additional boolean flag called \texttt{validPredecessor} is used to mark whether the node has a valid predecessor inside the sample space. This boolean is set to false if any of the predecessors of the node are set as invalid during the Trace analysis phase. This is done when running the \texttt{addNode} function over the trace which adds a new node to the CDAG.

With this set of nodes, we try to find a convex set of nodes to build the sampled CDAG. The motivation behind finding a convex set is so that we do not miss out on a dependence which went outside the sample space and came back in again. This ensures that all the dependencies are accounted for while calculating the upper bound cost. We will highlight this with an example, assuming two valid nodes in the CDAG \( V_1, V_2 \). \( V_1 \) has an outgoing edge to an invalid node \( V_3 \). And \( V_3 \) has an outgoing edge to \( V_2 \), we cannot ignore \( V_2 \) indirect dependence on \( V_1 \).

The following algorithm is used to calculate the convex set of nodes. We will start with an initial set of nodes, which we can be sure are a part of the convex set. The set of nodes inside the sample region which do not have any predecessors and which have validPredecessor set to TRUE are taken to be the initial set. We need the additional check of validPredecessor to ensure that nodes having an invalid predecessor, from non sampled space, are not counted in the convex set.

We can be sure that the set is convex, as all of these nodes do not have dependencies on any other prior nodes. We will refer to this convex set as \( C \).

To summarize the changes made
1. During the trace generation phase, memory addresses being written into by events happening outside the sample space are marked as invalid. These flags are used to mark nodes using these memory addresses as operands as invalid.

2. During the trace analysis, if a node being created had any invalid predecessors, then validPredecessor for the node would be set as FALSE.

3. We are going to generate the convex set using nodes which are inside the sample space and have their predecessors inside the sample space as well. That is why the initial set $C$ is the set of nodes with no predecessors and validPredecessor is set to TRUE. These nodes form a convex set as we can be sure that they do not have any incoming dependencies. Note that this is true only when sampling starts from beginning of the loop iteration.

4. Then we run the convex set generation algorithm on this initial set $C$ to add more nodes to the convex set.

   Additionally, each node has a dynamically calculated property numPreds, which is the number of predecessors of the node inside the sample space. $V$ is the set of nodes inside the sampled space.

   In the algorithm, we keep a visit count of number of incoming edges to a node from inside the sample space. If the number of incoming edges is equal to the number of predecessors of the node and if its validPredecessor is set to TRUE, then we can be sure that the node was visited from other nodes within the convex set and did not have an edge from an invalid node outside the sampled space.

   Then, this node is added to the list of nodes inside the queue. This queue is used to do the traversal to other successor nodes which are inside the sample space. We
Algorithm 1 Convex set generation

\begin{algorithm}
\begin{algorithmic}
\State $v.cnt \leftarrow 0 \ \forall \ v \in V$
\State $q.push(v) \ \forall \ v \in C$
\While{(!q.empty())}
\State $v \leftarrow q.pop()$
\For{each $s = successor(v)$}
\State $s.cnt++$
\If{$s.cnt == s.numPreds \&\& s.validPredecessor == TRUE$}
\State $q.push(s)$
\State addToConvexSet(s)
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

also add the node to the list of nodes forming the convex set. We can be sure that no node with an incoming edge from an invalid predecessor is added to the set.

Proof of correctness

As long as we start with a set of initial nodes which we can be sure are not invalid, then the proof is as follows. Assuming that the convex set C has a node v which has an edge from a node not in the set C. Since we are checking for the visit count from only nodes inside the set C, this node would not be added to the set C until the visit count is equal to the number of predecessors of this node. Then since we are adding 'v' to 'C' only if we can reach it through all the edges from the nodes in 'C', there cannot be any non-convex path.

3.1.3 Graph construction and heuristics

Once we have the list of nodes, we can use these to construct the new CDAG. The dynamic analysis tool has an option to generate the CDAG using a file. We follow these steps to obtain the sampled CDAG
1. Prune the node information to ensure that no invalid vertices are a part of the predecessor or successor list inherited from the original CDAG

2. Insert the updated convex set nodes along with the additional information like address, successor list, predecessor list to a file

3. Generate the new CDAG using the build CDAG API from a file

4. Run partitioning heuristics and cost analysis on the sampled CDAG

## 3.2 Evaluation and results

We used our automated sampling approach for analysis of standard benchmarks like jacobi-3d and benchmarks simulating real-world applications like GemsFDTD, Bwaves SPEC Benchmarks. [1]

### 3.2.1 Regular Benchmarks

In this section we will be comparing the results of sampling approach with the non-sampling approach. The cache size for which the cache miss cost is being calculated is along the X axis and the Y axis is the ratio of Compute Nodes in the program to cost of the schedule. Furthest in future cache policy is used to calculate the cost.

**Jacobi-2d**

The new schedule generated using the sampling approach shows that improvement can be made to the Jacobi-2d program. Jacobi-2d is stencil computation whose dependencies follow a regular pattern. We do see that a higher compute node to cost ratio is seen in the sampled approach. This could be due to cache size coming closer
Figure 3.1: Jacobi-2d sampled vs non sampled upper bound analysis.

to the number of total nodes in the sample CDAG.

**Jacobi-3d**

We see that new schedule shows improvement as compared to the regular schedule in both the sampling and non sampling approaches. Jacobi-3d is a stencil computation whose dependencies follow a regular pattern similar to Jacobi-2d discussed earlier.

### 3.2.2 SPEC Benchmarks

The automated sampling approach required identifying the hot loops in the program, this was done using HPCToolKit. Once the hot regions were annotated with
"ddg_start_trace" and "ddg_stop_trace" correctly, we ran the heuristics on the benchmarks and the results are evaluated below. Tile size is the initial tile used for convex Partitioning of the CDAG. The Y axes in the figure 3.3 is the ratio of compute nodes to total cost.

**GemsFDTD**

Gems shows that considerable improvement is possible with the use of tiling or other transformations to enhance data locality. On examining the code, we saw that it consists of a time iteration loop over a number of stencil computations. One thing to note here is that the GemsFDTD code has a regular pattern of dependencies. This makes it easier to sample the space and derive useful insights into the program.
We develop a strategy to improve the performance of this code using time tiling which is discussed in the next Chapter.

**Bwaves**

For the Bwaves benchmark our approach is unable to come up with a better schedule for the program. On a closer look at the code, we observe that there is a global reduction operation within an iterative loop, which makes tiling across the outer loop impossible. The approach correctly provided the tight upper bound for this benchmark.
Figure 3.4: Bwaves sampled upper bound analysis.
Chapter 4: Time Loop tiling

4.1 Introduction

With the upper bound analysis for GemsFDTD demonstrating that considerable improvement is possible, we decided to manually improve the benchmark and check its performance. We used OpenMP tasks to implement a time loop tiling strategy. With this strategy, all the loop nests inside the time loop were tiled and then defined as tasks. The next few sections give a background on OpenMP tasks, implementation details for Jacobi-2d and Gems FDTD and the results for this time tiling strategy.

4.2 OpenMP tasks

OpenMP task [10] construct allows you to define data dependencies for a task. When the task is going to be executed, these dependencies are checked and if the dependencies are satisfied then the task is executed. Three types of dependence types can be declared with tasks

1. Output dependence type(out): This task would be executed before any tasks using the address in output dependence is executed.

2. Input dependence type(in): Previous tasks with this address as output dependencies would be executed first before executing the current task.
3. Input and output dependence type (inout): Both input and output dependencies are specified on the given address.

Task scheduler creates a dynamic dependence graph and adds newly created tasks to it. At the same time, the scheduler is able to allocate available tasks to threads which get freed up.

Advantages of using tasks to create fine-grained parallelism are threefold

1. Achieve time loop tiling

2. Exploit data locality inherent with running multiple iterations of same tile across the time loop

3. Load balancing of work shared between threads is done by the task scheduler

4.3 Strategy and implementation

Firstly all the loop nests inside the time loop were tiled. A single master thread iterates through the time loop and creates tasks. Other threads are allocated these new tasks when the data dependencies defined in the tasks are satisfied. We faced one difficulty while defining the dependencies using the following pragma

\$omp task depend(in: A[i:range][j:range])

The range defined on an index was not being checked for the data dependencies, instead only the first element was being checked. The OpenMP specifications[10, p. 66] mentioned this feature but it still has not been implemented in gcc.

We resolved this issue by using the same tile ranges for all the loop nests and comparing the first element in each tile whenever defining dependence checks. Figure 4.1 makes it clearer. When dependencies for first tile are being checked, (0,0) element
in each tile is checked against the same element in other tiles.

We implemented the strategy for Jacobi-2D and GemsFDTD. The results from the experiments and few observations are described in the following sections.

4.3.1 Jacobi-2D

We compared sequential run-time with our OpenMP tasks strategy and an OpenMP for strategy.

Jacobi-2D has two computation steps inside the time loop. These are executed using 2d loop nests. We tiled the loop nests and generated tasks for each of the tiles. The dependencies are generated for each of the tiles above, below, to the left, and right of the current tile being executed. These were captured using the ’depend’ clause. A snippet of the code is copied below to show how the dependencies were defined.
Listing 4.1: Jacobi2d task dependence code snippet

```c
// block size
for (i=0; i<n; i+=b)
    for (j=0; j<n; j+=b)
    {
        i_left = (i==0)? 0: (i-b);
        j_bot = (j==0) ? 0: (j-b);
        i_right = ((i+b) >=n)?i:i+b;
        j_top = ((j+b) >= n) ? j : j+b;
        #pragma omp task
        firstprivate(t,i,j,i_left,i_right,j_bot,j_top,n,b)
        depend(in:A[i_left:b][j:b], A[i_right:b][j:b],
               A[i:b][j_bot:b],A[i:b][j_top:b],A[i:b][j:b],)
        depend(out:B[i:b][j:b])
        {
            for(int ii=i; ii<i+b; ii++)
                for(int jj=j; jj<j+b; jj++)
                {
                    if ((ii!=0 && ii!=n-1) && (jj!=0 && jj!=n-1))
                                            A[1+ii][jj] + A[ii-1][jj]);
                }
        }
    }
```

Figure 4.2 shows the various run-times as a function of tile size used to tile the loops.

We observe a few trends in figure 4.2. With a small tile size, the number of tasks getting created are high, that is why the initial overhead is observed for OpenMP tasks method. With OpenMP for strategy, higher parallelism can be obtained at small tile sizes which tends towards the sequential run, as tile size tends towards 1024, which is the size of the problem. Best performance for the OpenMP task strategy can be seen...
Figure 4.2: Comparison of sequential, OpenMP tasks, OpenMP for implementation of Jacobi-2d.
when the overhead of tasks creation overlaps with the computation inside the task. We believe this is the optimum number of tasks vs tile size balance which we should be aiming for.

**GemsFDTD**

Following steps were taken to implement the OpenMP tasks strategy in GemsFDTD code.

1. Inlined the code. Functions from UPML.F90, Update.f90, NFT.f90 and Huygens.f90 were inlined along with declaring the necessary variables as public.

2. All the loop nests were tiled with a universal tiling strategy described earlier. In certain places in the program, where array slicing was being used, transformations were used to change the code to nested for loops so as to enable tiling.

3. Tasks were defined for all the tiles. The following example highlights how the dependencies were declared for a small 2d nested loop. Merge is a ternary operator in Fortran[7]. The dependencies exist from current tile to the next tile in each dimension in this example.

```
Listing 4.2: GemsFDTD task dependence code snippet

```

```fortran
  do k=1,nz,block
    do j=1,ny,block
      do i=1,nx,block
        i_dep = merge(i,i+block,i+block>nx)
        j_dep = merge(j,j+block,j+block>ny)
        k_dep = merge(k,k+block,k+block>nz)
        !$omp task firstprivate(ts,i, j, k, ii, jj, kk) &
        !$omp& firstprivate(i_dep, j_dep, k_dep) &
        !$omp& depend(in:Ex(i,j,k), Ex(i,j,k_dep), Ex(i,j_dep,k)) &
```
Figure 4.3 was obtained by comparing the sequential tiled run with the OpenMP tasks strategy. The graph compares the improvement seen with increasing number of time loop iterations. OpenMP task strategy is able to perform better than the sequential run. We believe this shows the viability of the strategy. Future direction of the possible ways to further optimize this are discussed in Chapter 6.
Figure 4.3: Comparison of sequential, OpenMP tasks implementation of GemsFDTD.
Chapter 5: Conclusion

The DDG dynamic analysis framework is able to measure the bounds on data movement costs and give insights on application optimization. This empowers the application developer to further improve the performance and energy efficiency of his application. However, the upper bound analysis tool requires large disk space, creates a large DDG which requires huge memory and the heuristics take a long time to run. We put forth a sampling approach to upper bounds analysis which is able to give us similar insights into an application’s performance. The caveat is that the application should have regular dependencies so as to allow the sampled region to capture similar trends as seen in the full iteration space.

We developed and implemented the sampling approach as an extension of the DDG analysis framework and tested its effectiveness with the non sampling approach. The trends observed for Jacobi-2d and Jacobi-3d were similar to the trends observed with the non sampling approach. We tested the sampling approach for real world applications simulated by SPEC benchmarks as well. The experiments for Bwaves were able to show that the schedule cannot be improved for Bwaves but in the case of GemsFDTD it showed room for improvement.

The second part of the thesis puts forward a time loop tiling strategy which was implemented in Jacobi-2d and GemsFDTD. The strategy leverages the OpenMP
tasks construct to tile across the time loop by defining explicity dependencies on each task tile. The strategy shows considerable improvement over sequential runs and is comparable to the OpenMP for implementation. The strategy can be further optimized by attaching tasks to specific processors. This is explained in Chapter 6.
Chapter 6: Future Work

The DDG dynamic analysis framework is a strong tool to measure the data movement costs of an application. Upper bound analysis tool can be improved by capturing anti and output dependencies as well which would allow us to come up with a stronger upper bound schedule. The sampling approach does some over-approximations when creating the sampled CDAG. This process can definitely be more streamlined and improved during the CDAG creation phase.

Sampling approach can be improved by implementing the sample start and stop to be anywhere in the area of interest rather than the beginning of the iterations. This can be achieved by making a few additional changes mentioned below

1. We can check if the nodes executed occur in a lexicographical order earlier than the sample start point. These nodes should not invalidate any nodes in our sample space.

2. Initial set of nodes to start the convex set generation from should include nodes who had all the predecessors executed lexicographically before the sampling started and have their validPredecessor flag set to TRUE.

This way we can run multiple samples across the iteration space to obtain a better estimate on the upper bounds trend.
The time loop tiling OpenMP tasks strategy can be optimized in a few ways mentioned below

1. Processor affinity can be used to tie task tiles covering the same iteration space to the same processor to improve the performance.

2. In GemsFDTD implementation of the tasks which compute the boundaries can be fused with the task tiles to reduce the total number of tasks being generated and improve the load balance of the task tiles.

3. Task priority can be set to ensure that the tasks which are identified as critical are executed with a higher priority. This could be useful in GemsFDTD, more specifically with the single task which is used to reset the values of certain elements in Ex and Hx to 0.
Appendix A: Running the DDG tool for the sampled space

The following steps are used to create the sampled CDAG for a Jacobi program (example).

DDG_PREFIX= <Location of DDG install>
clang -c -flto -o jacobi1d.bc jacobi1d.c
opt -S -load ${DDG_PREFIX}/lib/ddg-instr.so -mem2reg -indvars -instrument-ddg
   -instrument-indvars -sample-start=0 -sample-stop=9
   jacobi1d.bc -o jacobi1d.instr.bc
clang -L ${DDG_PREFIX}/lib/ -lddg-rt jacobi1d.instr.bc

Once the trace has been generated the upper bound heuristics tool can be run on this sampled graph.
Bibliography


36