Techniques for Characterizing the Data Movement Complexity of Computations

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Venmugil Elango,
Graduate Program in Computer Science and Engineering

The Ohio State University
2016

Dissertation Committee:
Prof. P. Sadayappan, Advisor
Dr. Fabrice Rastello, Co-Advisor
Prof. Atanas Rountev
Prof. Radu Teodorescu
© Copyright by
Venmugil Elango
2016
The execution cost of a program, both in terms of time and energy, comprises computational cost and data movement cost (e.g., cost of transferring data between CPU and memory devices, between parallel processors, etc.). Technology trends will cause data movement to account for the majority of energy expenditure and execution time on emerging computers. Therefore, computational complexity alone will no longer be a sufficient metric for comparing algorithms, and a fundamental characterization of data movement complexity will be increasingly important.

In their seminal work, Hong & Kung proposed the red-blue pebble game to model the data movement complexity of algorithms. Using the pebble game abstraction, Hong & Kung proved tight asymptotic lower bounds for the data movement complexity of several algorithms by reformulating the problem as a graph partitioning problem. In this dissertation, we develop a novel alternate graph min-cut based lower bounding technique. Using our technique, we derive tight lower bounds for different algorithms, with upper bounds matching within a constant factor. Further, we develop a dynamic analysis based automated heuristic for our technique, which enables automatic analysis of arbitrary computations. We provide several use cases for our automated approach.

This dissertation also presents a technique, built upon the ideas of Christ et al. [15], to derive asymptotic parametric lower bounds for a sub-class of computations, called
affine computations. A static analysis based heuristic to automatically derive parametric lower bounds for affine parts of the computations is also presented.

Motivated by the emerging interest in large scale parallel systems with interconnection networks and hierarchical caches with varying bandwidths at different levels, we extend the pebble game model to parallel system architecture to characterize the data movement requirements in large scale parallel computers. We provide interesting insights on architectural bottlenecks that limit the performance of algorithms on these parallel machines.

Finally, using data movement complexity analysis, in conjunction with the roofline model for performance bounds, we perform an algorithm-architecture codesign exploration across an architectural design space. We model the maximal achievable performance and energy efficiency of different algorithms for a given VLSI technology, considering different architectural parameters.
I would like to start with thanking my advisor Prof. P. Sadayappan, without whose guidance this work would have been impossible. I am always amazed by the enthusiasm and energy with which he approaches and discusses new ideas. Suggestions and ideas that he provided me during our discussions over the past five years have proved to be invaluable to me.

I am grateful to my co-advisor, Dr. Fabrice Rastello. I learned a lot from his novel ideas and insightful comments. I admire his close attention to minor technical details and correctness of the ideas.

I extend my gratitude to Dr. Louis-Noël Pouchet for his helpful suggestions on various parts of this work. Interactions with him, especially with regards to experiments and presentation of ideas, have been invaluable.

I would like to thank Prof. J. Ramanujam for all the help he provided over the course of this work. Useful discussions with him, especially when we were close to various paper deadlines, were of immense help.

I extend my thanks to Prof. Nasko Rountev for his helpful suggestions and feedbacks on our work related to dynamic analysis, and on the presentation of our work in various papers and talks.

I would also remember interesting conversations I had with my labmates Martin Kong, Naser Sedaghati, Mahesh Ravishankar, Kevin Stock, Naznin Fauzia, Sanket
Tavargeri, Samyam Rajbhandari, Prashant Rawat, Wenlei Bao and Changwan Hong during my period of stay at DL574.

My friends Nithin, Manojprasadh, Saktheesh, Aravind, Ashok and Nikilesh extended their help and support during the times I needed them the most. I will always cherish the memorable experiences I had with them.

Finally, I am thankful to my parents and my family for all their support.
VITA

Jul 2003 – May 2008 .......................Bachelor of Engineering
Mechanical Engineering (Sandwich)
PSG College of Technology
Coimbatore, India.

Jun 2008 – Aug 2010 .......................Engineer
GE Aviation
Bangalore, India.

Dec 2010 – Present .........................Graduate Research Assistant
The Ohio State University
Columbus, Ohio.

May 2013 – Aug 2013 .......................Intern
Los Alamos National Laboratory
Los Alamos, New Mexico.

May 2015 – Aug 2015 .......................Intern
Lawrence Livermore National Laboratory
Livermore, California.

PUBLICATIONS

Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan
On Characterizing the Data Access Complexity of Programs.
In *ACM Symposium on Principles of Programming Languages*, January 2015.

Venmugil Elango, Naser Sedaghati, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, Radu Teodorescu, P. Sadayappan
On Using the Roofline Model with Lower Bounds on Data Movement.
Venmugil Elango, Fabrice Rastello, Louis-Noël Pouchet, J. Ramanujam, P. Sadayappan
On Characterizing the Data Movement Complexity of Computational DAGs for Parallel Execution.
In ACM Symposium on Parallelism in Algorithms and Architectures, June 2014.

Naznin Fauzia, Venmugil Elango, Mahesh Ravishankar, J. Ramanujam, Fabrice Rastello, Atanas Rountev, Louis-Noël Pouchet, P. Sadayappan
Beyond reuse distance analysis: Dynamic analysis for characterization of data locality potential.

Mahesh Ravishankar, Roshan Dathathri, Venmugil Elango, Louis-Noël Pouchet, J. Ramanujam, Atanas Rountev, P. Sadayappan
Distributed memory code generation for mixed Irregular/Regular computations.
In ACM Symposium on Principles and Practice of Parallel Programming, February 2015.

Pai-Wei Lai, Humayun Arafat, Venmugil Elango, P. Sadayappan
Accelerating Strassen-Winograd’s matrix multiplication algorithm on GPUs.

FIELDS OF STUDY

Major Field: Computer Science and Engineering

Studies in High Performance Computing: Prof. P. Sadayappan
# 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>vi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xi</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xiv</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xv</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xvi</td>
</tr>
<tr>
<td>Chapters:</td>
<td></td>
</tr>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Data Movement Complexity</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Computation Directed Acyclic Graph</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Characterizing the Data Movement Complexity of Computations</td>
<td>6</td>
</tr>
<tr>
<td>1.4 Outline</td>
<td>9</td>
</tr>
<tr>
<td>2. Red-Blue Pebble Game</td>
<td>11</td>
</tr>
<tr>
<td>2.1 Background</td>
<td>12</td>
</tr>
<tr>
<td>2.2 Red-Blue Pebble Game Without Recomputation</td>
<td>14</td>
</tr>
<tr>
<td>2.3 Decomposition</td>
<td>15</td>
</tr>
<tr>
<td>2.3.1 Disjoint Decomposition</td>
<td>15</td>
</tr>
<tr>
<td>2.3.2 Non-disjoint Decomposition</td>
<td>18</td>
</tr>
<tr>
<td>2.4 Input/Output Tagging/Untagging</td>
<td>22</td>
</tr>
<tr>
<td>2.5 Related Work</td>
<td>24</td>
</tr>
</tbody>
</table>

viii
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Energy trends</td>
<td>2</td>
</tr>
<tr>
<td>1.2</td>
<td>CDAG of single-sweep seidel algorithm. Input vertices are represented in black.</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Example illustrating decomposition of a composite program. Figure 2.1(a) shows an example code composed of four loop nests. Figure 2.1(b) shows the full CDAG corresponding to the four loop nests. Figure 2.1(c) shows a decomposed CDAG, where each sub-CDAG corresponds to two of the four loop nests.</td>
<td>16</td>
</tr>
<tr>
<td>3.1</td>
<td>Diamond DAG. Vertices shaded in black and grey represent input and output vertices, respectively.</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Diamond DAG with bow-tie shaped sub-DAG</td>
<td>33</td>
</tr>
<tr>
<td>4.1</td>
<td>Lower bound for Diamond DAG: Analytical versus automated CDAG analysis.</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Diamond-Chain CDAG [9].</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Lower bound for Diamond-Chain: Analytical versus automated CDAG analysis.</td>
<td>62</td>
</tr>
<tr>
<td>4.4</td>
<td>Lower bound for two-dimensional Jacobi computation: Analytical versus automated CDAG analysis.</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Quicksort versus odd-even sort: Normalized lower bound and cost for sequential execution order.</td>
<td>66</td>
</tr>
</tbody>
</table>
4.6 Execution time for original and tiled odd-even sort relative to quicksort. 67

4.7 Lower bounds of memory bandwidth requirement per FLOP for Jacobi and CG; and machine balance numbers for different processors. 69

5.1 Lower bound for matmult: Analytical versus automated CDAG analysis. 83

5.2 Comparison of data movement cost of Pluto’s register-tiled schedule against the lower bound. 84

5.3 Performance of register-tiled codes with Pluto’s original schedule and Pluto’s schedule with manual permutation applied. 86

6.1 Jacobi 1D computation, its CDAG and flow-dependence graph. 92

6.2 Illustration of Loomis-Whitney inequality in three-dimensional space. Here, volume of the red sphere is bounded from above by the product of square-root of the area of the three projected circles. 93

6.3 Illustration of Brascamp-Lieb inequality. 94

6.4 Example to illustrate the relationship between geometric inequality and data movement lower bounds. 96

6.5 Set of disjoint paths in the CDAG corresponding to the injective circuit $K_1$. 102

6.6 Iteration domain space for Jacobi 1D. Blue and red circles: Points representing dynamic instances of statements $S2$ and $S3$, respectively; Black arrows: Relation $R_{K_1}$ of injective circuit $(e_7,e_8)$; Black diamonds: Frontier $F_1$; Gray box: subset of points and corresponding vertex-set $U$. 104

6.7 Original and transformed spaces for Jacobi 1D example. 105

6.8 Broadcast path $P_1$ and its corresponding sub-CDAG. 108

7.1 Parallel machine model. 120

8.1 Roofline model: Performance and Energy rooflines. 138
8.2 Operational intensity upper bounds as a function of $\alpha$. 149
8.3 Upper bounds on performance for different choices of $\alpha$. 150
8.4 Upper bounds on performance. 151
8.5 Upper bounds on energy efficiency. 155
8.6 Upper bounds on energy efficiency 160
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Min-cut based lower bounds and corresponding upper bounds for different algorithms. (Lower order terms are ignored in the summary.)</td>
</tr>
<tr>
<td>4.1</td>
<td>Decomposability factors for different algorithms.</td>
</tr>
<tr>
<td>4.2</td>
<td>Ratios of lower bounds for unrestricted and restricted schedules for two SPEC benchmarks.</td>
</tr>
<tr>
<td>7.1</td>
<td>Specifications of computing systems.</td>
</tr>
<tr>
<td>8.1</td>
<td>List of symbols and their definitions.</td>
</tr>
<tr>
<td>8.2</td>
<td>Notations used in architecture design space exploration.</td>
</tr>
<tr>
<td>8.3</td>
<td>Nehalem-EX processor spec.</td>
</tr>
<tr>
<td>8.4</td>
<td>Nehalem-EX die dimensions.</td>
</tr>
<tr>
<td>8.5</td>
<td>Modeled LLC Slices using CACTI [38]</td>
</tr>
<tr>
<td>8.6</td>
<td>Nehalem-EX: effect of changing voltage and frequency on core/chip power</td>
</tr>
</tbody>
</table>
# LIST OF ALGORITHMS

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1 Conjugate Gradient method.</td>
<td>40</td>
</tr>
<tr>
<td>4.1 Algorithm to compute min-cut based LB of a sub-CDAG.</td>
<td>59</td>
</tr>
<tr>
<td>4.2 Algorithm to compute min-cut based LB of a CDAG through recursive</td>
<td>60</td>
</tr>
<tr>
<td>bisection.</td>
<td></td>
</tr>
<tr>
<td>4.3 Checks fuseability and tileability potential of loop nests.</td>
<td>72</td>
</tr>
<tr>
<td>4.4 Checks fuseability of two loop nests.</td>
<td>73</td>
</tr>
<tr>
<td>4.5 Checks tileability of a loop nest.</td>
<td>74</td>
</tr>
<tr>
<td>6.1 For each vertex $v$ in a flow-dependence graph $G_F$, finds a set of</td>
<td>112</td>
</tr>
<tr>
<td>paths and computes the corresponding complexity.</td>
<td></td>
</tr>
<tr>
<td>6.2 For a vertex $v$, try to add path $p$ to some other paths. Return</td>
<td>113</td>
</tr>
<tr>
<td>true if a good bound is found.</td>
<td></td>
</tr>
<tr>
<td>6.3 For a vertex $v$, selects a set of paths and computes the associated</td>
<td>113</td>
</tr>
<tr>
<td>complexity.</td>
<td></td>
</tr>
<tr>
<td>6.4 For a domain $D$ and a set of compatible subspaces, writes the linear</td>
<td>114</td>
</tr>
<tr>
<td>program, and returns the data movement lower bound (with cases).</td>
<td></td>
</tr>
</tbody>
</table>
# LIST OF LISTINGS

<table>
<thead>
<tr>
<th>Listing</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Untiled implementation of single-sweep seidel</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Tiled implementation of single-sweep seidel</td>
<td>3</td>
</tr>
<tr>
<td>5.1 Register-tiled code generated by Pluto</td>
<td>85</td>
</tr>
<tr>
<td>5.2 Register-tiled code generated by Pluto</td>
<td>86</td>
</tr>
<tr>
<td>6.1 Matrix-matrix multiplication</td>
<td>107</td>
</tr>
<tr>
<td>6.2 Matrix-matrix multiplication</td>
<td>115</td>
</tr>
<tr>
<td>6.3 Code with same array accesses as matrix-matrix multiplication</td>
<td>115</td>
</tr>
</tbody>
</table>
CHAPTER 1

Introduction

The execution cost of a program, both in terms of time and energy, is comprised of *computational cost* and *data movement cost* (e.g., cost of transferring data between CPU and memory devices, between parallel processors, etc.). Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational power of processors relative to the cost of memory access. The Intel 80286 processor introduced in 1982 had computational latency of 320 nanoseconds (ns) and memory access time of 225 ns [40]. A recent Intel Core i7 processor has a computational latency of 4ns and memory access latency of 37 ns, illustrating an order of magnitude shift in the ratio of computational cost to data movement cost. Further, the energy required for data movement is much higher than that required for performing an operation as summarized in Figure 1.1. The data shows the high energy cost of data movement relative to computation; this imbalance will only get worse with future technology generations [7, 22, 50]. Therefore, optimizing data movement costs will become ever more critical in the coming years. Given the crucial importance of optimizing data movement costs in systems, it is of significant and practical interest to characterize the inherent *data movement complexity* (a.k.a. I/O complexity) of algorithms.
1.1 Data Movement Complexity

A problem may be computed using different algorithms. For instance, the standard $O(N^3)$ algorithm and Strassen’s algorithm are two alternative algorithms for multiplying two matrices. An algorithm for a problem could have different implementations (e.g., untiled and tiled implementations of standard matrix multiplication). While different implementations of a particular algorithm have same the computational cost, their data movement costs may vary. This is illustrated using a simple example below.

Listing 1.1 shows an implementation of single-sweep seidel algorithm. Its computational cost can be simply stated as $(N-2)^2$ arithmetic operations. Listing 1.2 shows a functionally equivalent form of the same algorithm, after a tiling transformation of the code in Listing 1.1. The tiled form too has exactly the same computational cost.
of \((N-2)^2\) arithmetic operations. Next, let us consider the data movement cost for execution of these two implementations on a processor with a single level of cache. If the problem size \(N\) is larger than cache size \(S\), the number of cache misses would be higher for the untiled version (Listing 1.1) than the tiled version (Listing 1.2). But if the cache size were sufficiently large, the tiled version would not offer any benefits in reducing cache misses. Thus, unlike the computational complexity of an algorithm, which stays unchanged for different implementations and is also independent of machine parameters like cache size, the data movement cost depends both on the cache capacity and the order of execution of the operations (i.e., implementation) of the algorithm. The order of execution of the operations in an implementation is often referred as the \textit{schedule} of the implementation. A fundamental question therefore is:
Given an algorithm and the amount of storage at different levels of the cache/memory hierarchy, what is the minimum possible number of data transfers at the different levels, among all valid schedules that perform the operations?

The data movement complexity, or, I/O complexity of an algorithm is defined as the minimum data movement cost, over all possible valid schedules of the algorithm. A formal definition of data movement complexity is provided in Section 2.1. Valid schedules of an algorithm are the ones that respect the data dependence constraints of the algorithm. While, in general, it is intractable to precisely determine the absolute minimum number of data movements among all valid schedules of an algorithm, it is feasible to develop lower bounds on the optimal number of data movements, which is the focus of this dissertation.

The single-sweep seidel algorithm, used as our example above, has a matching lower and upper bounds, providing the asymptotic data movement complexity of \( \Theta(N^2/S) \), which is dependent on the parameters: problem size \( N \) and cache size \( S \). Its computational complexity on the other hand, which is dependent only on the problem size, is \( \Theta(N^2) \).

1.2 Computation Directed Acyclic Graph

In order to model the range of valid schedules for the operations of an algorithm for a given input, it is common to use the schedule-invariant abstraction of the computation directed acyclic graph (CDAG). Each vertex of a CDAG corresponds to either an input or an instance of a computation. Edges of a CDAG capture true data dependences from producer instances to consumer instances. We use a notation similar to that of Bilardi & Peserico [8] to formally describe a CDAG.
**Definition 1** (Computation Directed Acyclic Graph (CDAG)). A *computation directed acyclic graph* (CDAG) is a 4-tuple \( C = (I, V, E, O) \) of finite sets such that:

1. \( I \subseteq V \) is the input set with no incoming edges.

2. \( E \subseteq V \times V \) is the set of edges.

3. \( G = (V, E) \) is a directed acyclic graph with no isolated vertices.

4. \( V \setminus I \) is called the operation set.

5. \( O \subseteq V \) is called the output set.

Two important characteristics of this abstract form of representing a computation are that (1) there is no specification of any particular order of execution of the operations: the CDAG abstracts the schedule of operations by only specifying partial ordering constraints as edges in the graph; (2) there is no association of memory locations with the source operands or result of any operation.

![CDAG of single-sweep seidel algorithm. Input vertices are represented in black.](image_url)
Figure 1.2 shows the CDAG for the codes in Listing 1.1 and Listing 1.2, for input $N=6$. Although the schedules for tiled and untiled versions are different, the set of computational instances and the producer-consumer relationships for the flow of data are exactly the same.

1.3 Characterizing the Data Movement Complexity of Computations

In their seminal work [27], Hong & Kung formalized the problem of data movement complexity and proposed the *red-blue pebble game* to model the data movement complexity of algorithms. Hong & Kung formulated the problem of finding lower bounds for data movement complexity as a graph partitioning problem, called *S-partitioning*. In this dissertation, we develop new manual and automated techniques to derive lower bounds for data movement complexity of computations and present applications and use cases for our analysis.

**CDAG decomposition and input/output tagging:** Many practical applications are composed of several sub-computations. Using manual reasoning to directly analyze the data movement complexity of such composite applications is very difficult and may not even be feasible. It becomes much easier to manually derive the lower bounds for data movement complexities of such algorithms if there exists a way to reason on each sub-computation individually and obtain a valid consolidated result for the full computation. The first contribution of this dissertation involves developing a technique called **CDAG decomposition**, that allows to decompose CDAGs into sub-CDAGS (each corresponding to a sub-computation), derive lower bounds on
each sub-CDAG separately, and finally consolidate the results from sub-CDAGs to obtain valid lower bounds for the full CDAG.

Certain lower bounding techniques, such as Hong & Kung’s $S$-partitioning technique rely on the presence of inputs on the (sub-)CDAGs to obtain meaningful lower bounds. However, performing CDAG decomposition generally leaves sub-CDAGs with few inputs, leading to trivial bounds. In order to overcome this problem, we next develop a technique called input/output tagging/untagging, that allows us to “tag” some vertices of the (sub-)CDAG as inputs, derive meaningful bounds and finally make adjustments to derived bound to account for input/output tagging.

**Graph min-cut based lower bounding technique:** Following Hong & Kung’s work on $S$-partitioning to derive lower bounds, several methods have been proposed to derive tight bounds for various algorithms. Our next contribution involves developing a novel lower bounding technique based on graph min-cuts. We use our proposed technique to derive tight lower bounds on data movement complexity for different algorithms. Our method provides tight lower bounds for certain computations for which Hong & Kung’s technique provides trivial bounds. Our technique has an added benefit of being automatable. We develop methods to automate the process of deriving data movement lower bounds for arbitrary computations using our newly developed lower bounding method through dynamic analysis.

**Static analysis for affine computations:** Christ et al. [15] developed a novel approach based on geometric reasoning to automatically derive parametric lower bounds for affine computations under geometric model abstraction. The geometric model is an alternative abstraction for CDAG abstraction, but it a weaker model
than CDAG model in the sense that it ignores data dependences between operation instances. This gives rise to a few difficulties when using the geometric model to derive bounds for affine computations. In this dissertation, we build upon the work of Christ et al. [15] to develop methods to apply similar geometric reasoning on the CDAGs corresponding to affine parts of computations. This allows us to overcome some of the limitations associated with the analysis under geometric model. We also provide a static analysis based heuristic to automate this process of deriving parametric lower bounds for affine computations. A description of this work also appeared in [19].

**Parallel memory model:** We next extend the red-blue pebble game model to characterize data movements in large scale parallel machines. A similar extension has been proposed in the past for different parallel environments [47, 53]. In this work, our extension is targeted for multi-node/multi-core machines. With the help of our extended model, we provide analysis and interesting insights on architectural bottlenecks that limit the performance of algorithms. This work was also published in [18].

**Application to algorithm-architecture codesign:** Our final contribution involves an application of data movement lower bounds to perform algorithm-architecture codesign exploration across an architectural design space using **roofline model**. Previous attempts [55, 16, 13, 14] to analyze bottlenecks in performance of an architecture using the roofline model have performed the analysis considering specific implementations of algorithms. With the help of data movement lower bounds, we are able
to derive conclusions for an algorithm as a whole, independent of any specific implementation. We model the maximal achievable performance and energy efficiency of different algorithms for a given VLSI technology, considering different architectural parameters. A description of this work appeared in [20].

1.4 Outline

In Chapter 2, we introduce the red-blue pebble game abstraction proposed by Hong & Kung to characterize the data movement complexity of algorithms. We present a modification to Hong & Kung’s pebble game model to restrict recomputation and derive results needed to obtain tighter lower bounds in subsequent chapters when recomputation is disallowed.

In Chapter 3, we develop a novel graph min-cut based lower bounding technique. We derive lower bounds (that match the corresponding upper bounds within a constant factor) for different algorithms using our proposed technique.

Chapter 4 presents an automated approach that is based on the lower bounding technique developed in Chapter 3 to automatically derive bounds for arbitrary CDAGs. We demonstrate the effectiveness and several uses of our automated approach in Section 4.3.

Hong & Kung allowed recomputation in their model and developed $S$-partitioning technique for lower bounds using their pebble game model. In Chapter 5, we present an adaptation of Hong & Kung’s $S$-partitioning technique for the model that disallows recomputation. We also provide an integer linear programming based heuristic for automatically computing lower bounds based on $S$-partitions in a CDAG when recomputation is disallowed.
In Chapter 6, we build upon the ideas of [15] to derive parametric lower bounds for CDAGs that allows us to analyze a larger subset of algorithms than in [15]. We also provide a static analysis based heuristics to automate this process of deriving parametric lower bounds for affine parts of computations.

Chapter 7 details our extension of the pebble game model to parallel machines and its applications.

Application of the data movement complexity analysis to algorithm-architecture codesign exploration is presented in Chapter 8.

Finally, we draw conclusions from our current work and provide some possible directions for future work in Chapter 9.
CHAPTER 2

Red-Blue Pebble Game

Hong & Kung in their seminal work [27], proposed the red-blue pebble game to model the data movement complexity of algorithms. Using the pebble game formulation, Hong & Kung developed a graph partitioning based lower bounding technique, called S-partitioning, and derived a number of lower bound results for data movement requirements of several algorithms. In this chapter, we provide an overview of Hong & Kung’s red-blue pebble game in Section 2.1. Hong & Kung’s pebble game model implicitly allows recomputation, i.e., a single operation can be evaluated multiple times. However, several prior works have focused on deriving lower bounds in a restricted model that disallows recomputation in the interest of tractability for deriving tighter bounds. In Section 2.2, we describe a restricted model of the red-blue pebble game that disallows recomputation. In Section 2.3, we introduce the notion of decomposition of CDAGs. We heavily make use of this decomposition property (formally described in Theorem 1) in the subsequent chapters to derive tight lower bounds for different algorithms. Section 2.4 presents results that are specific to the model that prohibits recomputation. These results are necessary to obtain non-trivial lower bounds under certain circumstances.
2.1 Background

The red-blue pebble game, proposed in [27], is played on a CDAG. We are provided with two kinds of pebbles, namely, red and blue pebbles. Capacity of blue pebbles is considered to be unlimited, while red pebbles are of limited capacity, $S$. Thus, blue and red pebbles are analogous to slow and fast memories, respectively, in a serial system with two-levels of memory hierarchy. The rules of the red-blue pebble game closely depict the data movement operations in a two-level memory system.

Given a CDAG $C = (I, V, E, O)$ and $S$ red pebbles, the rules of the red-blue pebble game are as follows:

**R1 (Input)** A red pebble may be placed on any vertex that has a blue pebble (load from slow to fast memory).

**R2 (Output)** A blue pebble may be placed on any vertex that has a red pebble (store from fast to slow memory).

**R3 (Compute)** If all predecessors of a vertex $v \in V \setminus I$ have red pebbles, a red pebble may be placed on $v$ (execution of a computational operation).

**R4 (Delete)** A red pebble may be removed from any vertex (reuse storage).

Rules of Hong & Kung’s pebble game allow for **recomputation**, i.e., the rule R3 can be applied to a vertex $v$ multiple times.

**Definition 2** (Configuration). A configuration is defined as a pair $(V_1, V_2)$, where $V_1, V_2 \subseteq V$, such that $V_1$ comprises of vertices with red pebbles and $V_2$ comprises of vertices with blue pebbles. Vertices in $V_1 \cap V_2$ contain both red and blue pebbles.
An initial configuration of a CDAG $C = (I, V, E, O)$ is a configuration in which all the input vertices $I$ have blue pebbles. Similarly, a terminal configuration is one in which all the output vertices $O$ have blue pebbles on them.

**Definition 3** (Pebbling). Given a configuration $X_1$, pebbling is defined as application of one of the rules of a pebble game on a vertex $v$ of a CDAG, causing a change in the configuration $X_1$, leading to a new configuration $X_2$.

A pebbling by application a rule $R_x$ on a vertex $v$ is denoted by the notation $P(R_x, v)$. Pebbling a vertex $v$ using the rule $R_3$ (i.e., $P(R_3, v)$) is also informally referred to firing or computing $v$.

**Definition 4** (Pebbling sequence). Given a CDAG $C = (I, V, E, O)$ and $S$ red pebbles, a pebbling sequence is a sequence of pebblings that begins with the initial configuration and ends with the terminal configuration.

A subsequence of pebblings of a pebbling sequence is referred to as a pebbling subsequence.

**Definition 5** (Data movement cost of a pebbling sequence). The data movement cost of a pebbling sequence $P$ played with $S$ red pebbles, $q(P, S)$, is defined as the number of pebblings in $P$ according to the rules $R1$ or $R2$.

**Definition 6** (Optimal pebbling sequence). Given a CDAG $C$ and $S$ red pebbles, an optimal pebbling sequence is a pebbling sequence that achieves minimum data movement cost on $C$ when played $S$ red pebbles.

**Definition 7** (Data movement complexity). Data movement complexity of a CDAG $C$, $Q(C, S)$, is defined as the data movement cost of an optimal pebbling sequence for $C$ played with $S$ red pebbles.
When either $C$ or $S$ is clear from the context, we use the notation $Q(S)$ or $Q(C)$, respectively, to refer to $Q(C, S)$. Similarly, when both $C$ and $S$ are clear from the context, we simply use the notation $Q$ to refer to $Q(C, S)$.

**Remark 1.** Since a CDAG $C = (I, V, E, O)$ would have no isolated vertices and inputs have no incoming edges (see Definition 1), each vertex in the input set $I$ would have at least one successor. Hence, any pebbling sequence for $C$ will necessarily have at least one pebbling with rule R1 for each vertex in $I$. Similarly, any pebbling sequence for $C$ will have at least one pebbling with rule R2 for each vertex in the output set $O$.

### 2.2 Red-Blue Pebble Game Without Recomputation

A pebble game model that disallows recomputation can be formalized by changing the rule R3 of the red-blue pebble game to $R3_{NR}$ (NR denotes No-Recomputation) as follows:

**$R3_{NR}$ (Compute)** If all predecessors of a vertex $v \in V\setminus I$ have red pebbles and a red pebble has not been previously placed on $v$, a red pebble may be placed on (or moved to) $v$ (execution of a computational operation without recomputation).

Enforcing restriction on recomputation allows us to derive some results in Section 2.4 (which are invalid under the model with recomputation) that help us obtain tighter lower bounds in the later chapters. We also use the abbreviations $RB$ and $RB_{NR}$ to refer to red-blue pebble games with and without recomputation, respectively.

$^1$The original red-blue pebble game in [27] does not allow moving/sliding a red pebble from a predecessor vertex to a successor; we chose to allow it since it reflects real instruction set architectures. Others [45] have also considered a similar modification. All our proofs hold for both the variants.
2.3 Decomposition

Applications are typically constructed from a number of sub-computations using the fundamental composition mechanisms of sequencing, iteration and recursion. In the case of computational complexity, the complexity of the whole program can be obtained through simple addition of computational complexities of its sub-computations. A similar mechanism for the data movement complexity would be of interest to simplify the process of deriving data movement lower bounds for composite applications. Figure 2.1 illustrates this point.

Further, certain lower bounding techniques, such as our mincut based technique introduced in Chapter 3, heavily rely on ability to decompose a CDAG to derive tighter bounds.

This section derives necessary results that enable us to obtain data movement lower bounds of a composite program through simple addition of the lower bounds of its sub-computations. We would like to note that the decompositions results derived in this section are applicable to the pebble game models both with and without recomputation.

2.3.1 Disjoint Decomposition

Theorem 1 allows us to obtain the lower bound for a CDAG through direct addition of lower bounds of its decomposed sub-CDAGs.

**Theorem 1 (Decomposition).** Given a CDAG $C = (I, V, E, O)$, let $\{V_1, V_2, \ldots, V_n\}$ be an arbitrary (not necessarily acyclic) pairwise-disjoint partitioning of $V$ (i.e., $i \neq j \implies V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^n V_i = V$) and $C_1, C_2, \ldots, C_n$ be the induced sub-CDAGs (i.e., $I_i = I \cap V_i$, $E_i = E \cap V_i \times V_i$, $O_i = O \cap V_i$). If $Q(C)$ is the data
\textbf{Figure 2.1:} Example illustrating decomposition of a composite program. Figure 2.1(a) shows an example code composed of four loop nests. Figure 2.1(b) shows the full CDAG corresponding to the four loop nests. Figure 2.1(c) shows a decomposed CDAG, where each sub-CDAG corresponds to two of the four loop nests.

\textit{movement complexity of $C$ and $Q(C_i)$ is the data movement complexity of $C_i$, then $\sum_{i=1}^{n} Q(C_i) \leq Q(C)$. In particular, if $LB(C_i)$ is a data movement lower bound for $C_i$, then $\sum_{i=1}^{n} LB(C_i)$ is a valid data movement lower bound for $C$.}

\textit{Proof.} Consider an optimal pebbling sequence $\mathcal{P}$ (following the rules of red-blue pebble game with or without recomputation) for $C$, with cost $Q(C)$. We define the cost of $\mathcal{P}$ restricted to $V_i$, denoted as $Q(C)_{|V_i}$, as the number of R1 or R2 pebblings in $\mathcal{P}$ that involve a vertex of $V_i$. Clearly $Q(C) = \sum_{i=1}^{n} Q(C)_{|V_i}$. We will show that we can
build from $\mathcal{P}$, a pebbling sequence $\mathcal{P}_i$ for $C_i$, of cost $Q(C)|_{V_i}$. This will prove that $Q(C_i) \leq Q(C)|_{V_i}$, and thus $\sum_{i=1}^{n} Q(C_i) \leq \sum_{i=1}^{n} Q(C)|_{V_i} = Q(C)$.

$\mathcal{P}_i$, with cost $Q(C)|_{V_i}$, is built from $\mathcal{P}$ as follows: (1) Start with $\mathcal{P}_i = \mathcal{P}$; (2) for any pebbling in $\mathcal{P}_i$ that involves a vertex $v \in V_i$, leave this pebbling as such in $\mathcal{P}_i$; (3) delete all other pebblings in $\mathcal{P}_i$. Pebblings in $\mathcal{P}_i$ with rules R1, R2, and R4 trivially satisfy the rules of the game. Whenever a pebbling with rule R3 (or R3_{NR}) on a vertex $v$ is performed in $\mathcal{P}$, all the predecessors of $v$ must have had a red pebble on them. Since all pebblings of $\mathcal{P}$ on the vertices of $V_i$ are maintained in $\mathcal{P}_i$, when $v$ is fired in $\mathcal{P}_i$, all its predecessor vertices will have red pebbles, satisfying the conditions of rule R3 (or R3_{NR}). □

The following corollary directly follows from the decomposition theorem.

**Corollary 1** (Input/output separation). Given a CDAG $C = (I, V, E, O)$ and some set $dI \subseteq I$ and $dO \subseteq O$, let $I' = I \setminus dI$, $O' = O \setminus dO$, $V' = V \setminus (dI \cup dO)$, $E' = E \cap V' \times V'$ and $C' = (I', V', E', O')$. If $Q(C)$ and $Q(C')$ are the data movement complexities of $C$ and $C'$, respectively, then,

$$Q(C) \geq Q(C') + |dI| + |dO|$$

Proof. Let $C_1 = (dI, dI, \emptyset, \emptyset)$, $C_2 = (\emptyset, dO, E \cap dO \times dO, dO)$. Note that the set ${V', dI, dO}$ forms a pairwise disjoint partition of $V$ and, $C'$, $C_1$ and $C_2$ are the induced CDAGs of $C$. Let $Q(C_1)$ and $Q(C_2)$ be the data movement complexities of $C_1$ and $C_2$, respectively. Clearly $Q(C_1) \geq |dI|$ and $Q(C_2) \geq |dO|$. (See Remark 1.) Hence, from Theorem 1,

$$Q(C) \geq Q(C') + Q(C_1) + Q(C_2) \geq Q(C') + |dI| + |dO|. \quad \square$$
2.3.2 Non-disjoint Decomposition

The decomposition theorem in the previous subsection is applicable only when the decomposed sub-CDAGs of a CDAG $C$ are pairwise disjoint. Since some edges are removed during decomposition, in some cases, the resulting lower bound obtained using Theorem 1 might be weak. It might be possible to reduce this loss through careful non-disjoint decomposition (while maintaining the validity of derived bounds), so that certain critical dependences needed for deriving tighter bounds are maintained in the sub-CDAGs. As seen later in Section 3.3.4, there are many practical applications that benefit from a careful non-disjoint decomposition, which allows us to derive bounds with tighter multiplicative constants. Theorem 2 derives results for a specific kind of non-disjoint decomposition of CDAGs.

Before we state the non-disjoint decomposition theorem, we introduce the needed definitions below.

**Definition 8** (Ancestor-set). *Ancestor-set of a vertex $v \in V$, $\text{Anc}(v)$, in a DAG $G = (V, E)$ is a set of vertices that can reach $v$ through a non-empty directed path in $G$.*

**Definition 9** (Descendant-set). *Descendant-set of a vertex $v \in V$, $\text{Desc}(v)$, in a DAG $G = (V, E)$ is a set of vertices that can be reached from $v$ through a non-empty directed path in $G$.*

**Definition 10** (Non-disjoint bisection). *Given a CDAG $C = (I, V, E, O)$ and any vertex $x \in V$, we say $(C_1 = (I_1, V_1, E_1, O_1), C_2 = (I_2, V_2, E_2, O_2))$ is a non-disjoint bisection of $C$ w.r.t. $x$ if:*


1. $V_2 = \text{Desc}(x)$ and $C_2$ is the induced sub-CDAG of $C$, i.e., $I_2 = I \cap V_2$, $E_2 = E \cap V_2 \times V_2$ and $O_2 = O \cap V_2$.

2. $V_1 = V_{\text{rem}} \cup V_{\text{out}}$, where $V_{\text{rem}} = V \setminus V_2$ and $V_{\text{out}} = \{v | v \in V_2 \land (\exists u \in V_{\text{rem}}, (u, v) \in E)\}$; $E_1 = E_{\text{rem}} \cup E_{\text{out}}$, where $E_{\text{rem}} = E \cap V_{\text{rem}} \times V_{\text{rem}}$ and $E_{\text{out}} = \{x\} \times V_{\text{out}}$; $I_1 = I \cap V_1$; and $O_1 = O \cap V_1$.

Note that the vertices in $V_{\text{out}}$ are part of both the sub-CDAGs $C_1$ and $C_2$, and any dependence edges to (resp. from) vertices in $V_{\text{out}}$ in $C$ are maintained in $C_1$ (resp. $C_2$) after decomposition.

**Definition 11** (Non-disjoint decomposition). Given a CDAG $C = (I, V, E, O)$ and a set of vertices $X = \{x_1, x_2, \ldots, x_n\} \subset V$ such that $x_i \in \text{Desc}(x_{i-1})$, for $i = 2, 3, \ldots, n$, we say $C = (C_1, C_2, \ldots, C_{n+1})$, is a non-disjoint decomposition of $C$, if $C$ satisfies the following:

1. $(C_1, C_{x_1})$ is the non-disjoint bisection of $C$ w.r.t. $x_1$.

2. For $i = 2, 3, \ldots, n$, $(C_i, C_{x_i})$ is the non-disjoint bisection of $C_{x_{i-1}}$ w.r.t. $x_i$.

3. $C_{n+1} = C_{x_n}$.

The following Lemma allows us to bound the data movement complexity of a CDAG $C$ in terms of the data movement complexities of its non-disjoint bisection $(C_1, C_2)$.

**Lemma 1** (Non-disjoint bisection). Given a CDAG $C = (I, V, E, O)$ and any vertex $x \in V$, let $(C_1 = (I_1, V_1, E_1, O_1), C_2 = (I_2, V_2, E_2, O_2))$ be the non-disjoint bisection of $C$ w.r.t. $x$ as defined in Definition 10. If $Q(C, S)$ is the data movement complexity of $C$ with $S$ red pebbles, and $Q(C_1, S+1)$ and $Q(C_2, S)$ are the data movement complexities
of $C_1$ and $C_2$ with $S+1$ and $S$ red pebbles, respectively, then $Q(C_1, S+1) + Q(C_2, S) \leq Q(C, S)$. In particular, if $LB(C_1, S+1)$ and $LB(C_2, S)$ are data movement lower bounds for $C_1$ and $C_2$ with $S+1$ and $S$ red pebbles, respectively, then $LB(C_1, S+1) + LB(C_2, S)$ is a valid data movement lower bound for $C$ with $S$ red pebbles.

Proof. As in Definition 10, let $V_{rem} = V \setminus V_2$, $E_{rem} = E \cap V_{rem} \times V_{rem}$, $V_{out} = \{v | v \in V_2 \land (\exists u \in V_{rem}, (u, v) \in E)\}$ and $E_{out} = \{x\} \times V_{out}$.

Consider an optimal pebbling sequence $P$ (following the rules of red-blue pebble game with or without recomputation) for $C$, with cost $Q(C, S)$. We define the cost of $P$ restricted to $V_i$ when played with $S$ red pebbles, denoted as $Q(C, S)_{|_{V_i}}$, as the number of R1 or R2 pebblings in $P$ that involve a vertex of $V_i$. Clearly, $Q(C, S + 1)_{|_{V_{rem}}} + Q(C, S)_{|_{V_2}} = Q(C, S)$. We will show that we can build from $P$, a pebbling sequence $P_1$ for $C_1$ with $S+1$ red pebbles, of cost $Q(C, S + 1)_{|_{V_{rem}}}$, and a pebbling sequence $P_2$ for $C_2$ with $S$ red pebbles, of cost $Q(C, S)_{|_{V_2}}$. This will prove that $Q(C_1, S + 1) + Q(C_2, S) \leq Q(C, S + 1)_{|_{V_{rem}}} + Q(C, S)_{|_{V_2}} = Q$.

$P_1$, of cost $Q(C, S + 1)_{|_{V_{rem}}}$, is built from $P$ as follows: (1) Start by setting $P_1$ equal to the pebbling subsequence of $P$ restricted to $V_{rem}$; (2) For each vertex $v \in V_{out}$, successively append pebbling $P(R3, v)$ to $P_1$. This process of appending $P(R3, v)$ to $P_1$ is valid since, from the construction of $C_1$, all the vertices in $V_{out}$ have exactly one predecessor $x$ and we have one additional red pebble $s$. Hence at the end of step-(1), the additional pebble $s$ can be fixed on the vertex $x$ and rule R3 (or R3$_{NR}$) can be successively applied on each vertex $v \in V_{out}$ to fire $v$ with no additional data movement cost.

$P_2$, of cost $Q(C, S)_{|_{V_2}}$, is built from $P$ as follows: (1) Start with $P_2 = P$; (2) for any pebbling in $P_2$ that involves a vertex $v \in V_2$, leave this pebbling as such in $P_2$; (3)
delete all other pebblings in $\mathcal{P}_2$. Pebblings in $\mathcal{P}_2$ with rules R1, R2, and R4 trivially satisfy the rules of the game. Whenever a pebbling with rule R3 (or R3$_{NR}$) on a vertex $v$ is performed in $\mathcal{P}$, all the predecessors of $v$ must have had a red pebble on them. Since all pebblings of $\mathcal{P}$ on the vertices of $V_2$ are maintained in $\mathcal{P}_2$, when $v$ is fired in $\mathcal{P}_2$, all its predecessor vertices will have red pebbles, satisfying the conditions of rule R3 (or R3$_{NR}$). □

Finally, we prove the non-disjoint decomposition theorem below.

**Theorem 2** (Non-disjoint decomposition). Given a CDAG $C = (I, V, E, O)$ and a set of vertices $\{x_1, x_2, \ldots, x_n\} \subset V$ such that $x_i \in \text{Desc}(x_{i-1})$, for $i = 2, 3, \ldots, n$, let $(C_1, C_2, \ldots, C_{n+1})$ be the non-disjoint decomposition of $C$ as defined in Definition 11. If $Q(C, S)$ is the data movement complexity of $C$ with $S$ red pebbles, and $Q(C_i, S+1)$ is the data movement complexity of $C_i$, for $i = 0, 1, \ldots, n+1$, with $S+1$ red pebbles, then
\[
\sum_{i=1}^{n+1} Q(C_i, S+1) \leq Q(C, S).
\]
In particular, if $LB(C_i, S+1)$ is the data movement lower bound for $C_i$, where $i = 1, \ldots, n+1$, then
\[
\sum_{i=1}^{n+1} LB(C_i, S+1)
\]
is a valid data movement lower bound for $C$ with $S$ red pebbles.

**Proof.** This can be proved inductively as follows: For the base case, consider $(C_1, C_{x_1})$, the non-disjoint bisection $C$ w.r.t. $x_1$. From Lemma 1, $Q(C_1, S+1) + Q(C_{x_1}, S) \leq Q(C, S)$. Assume, as induction hypothesis, that the inequality
\[
\sum_{i=1}^{m} Q(C_i, S+1) + Q(C_{x_m}, S) \leq Q(C, S)
\] (2.1)
holds after $m$ steps, after application of $m$ non-disjoint bi-sectionings. At step $m+1$, $C_{x_m}$ is bisected into $(C_{m+1}, C_{x_{m+1}})$ through non-disjoint bi-sectioning. From Lemma 1,
we get
\[ Q(C_{m+1}, S + 1) + Q(C_{x_{m+1}}, S) \leq Q(C_{x_m}, S) \quad (2.2) \]

From inequalities (2.1) and (2.2), we have
\[ \sum_{i=1}^{m+1} Q(C_i, S + 1) + Q(C_{x_{m+1}}, S) \leq Q(C, S). \]

Finally, at step \( n \), \( C_{x_n} \) is bisected into \( (C_n, C_{n+1}) \). By induction, at the end of \( n \) steps, we have
\[ \sum_{i=1}^{n+1} Q(C_i, S + 1) \leq \sum_{i=1}^{n} Q(C_i, S + 1) + Q(C_{n+1}, S) \leq Q(C, S). \quad \Box \]

### 2.4 Input/Output Tagging/Untagging

When a CDAG is decomposed into sub-CDAGs, some of the sub-CDAGs might end up with very few or no input vertices (e.g., refer Figure 2.1(c)). Some lower bounding techniques (for example, Hong & Kung’s \( S \)-partitioning technique) rely on the presence of input vertices in a (sub-)CDAG to derive tighter bounds. Hence, absence of input vertices in sub-CDAGs, as a result of decomposition, might lead to very weak bounds. This section shows ways to overcome this problem using a technique called **input/output tagging**. The idea is to tag the predecessor-free non-input vertices \( (dI) \) of a (sub-)CDAG as inputs, compute the lower bound for this “tagged” (sub-)CDAG and make adjustments to the result to obtain a valid lower bound for the original (sub-)CDAG.

The techniques of input/output (un)tagging, presented in this section are applicable only to the restricted pebble game model that disallows recomputation.

**Theorem 3** (Input/Output (Un)Tagging). Let \( C \) and \( C' \) be two CDAGs with the same DAG \( G = (V, E) \) such that \( C = (I, V, E, O) \) and \( C' = (I \cup dI, V, E, O \cup dO) \), where \( dI \) and \( dO \) are some subsets of \( V \) and \( dI \cap dO = \emptyset \). If \( Q(C) \) is the data
movement complexity of C and \( Q(C') \) is the data movement complexity of \( C' \) then, \( Q(C) \) can be bounded by \( Q(C') \) as follows (tagging):

\[
Q(C') - |dI| - |dO| \leq Q(C)
\]  

(2.3)

Reciprocally, \( Q(C') \) can be bounded by \( Q(C) \) as follows (untagging):

\[
Q(C) \leq Q(C')
\]  

(2.4)

**Proof.** Consider an optimal pebbling sequence \( \mathcal{P} \) for \( C \), of cost \( Q(C) \). To prove the inequality (2.3), we will build a pebbling sequence \( \mathcal{P}' \) for \( C' \), of cost no more than \( Q(C) + |dI| + |dO| \). This will prove that \( Q(C') \leq Q(C) + |dI| + |dO| \). We will build \( \mathcal{P}' \) from \( \mathcal{P} \) as follows: (1) Start by copying all the pebblings in \( \mathcal{P} \) to \( \mathcal{P}' \), i.e., \( \mathcal{P}' \leftarrow \mathcal{P} \); (2) for any tagged input vertex \( v \in dI \), the (only) firing operation involving \( v \), \( P_{R3NR}^v \) in \( \mathcal{P} \) is replaced with \( (\mathcal{P}')_{R1}^v \) in \( \mathcal{P}' \) with an additional cost of \( |dI| \); (3) for any tagged output vertex \( v \in dO \), the (only) firing operation involving \( v \), \( P_{R3NR}^v \) in \( \mathcal{P} \) is supplemented with pebbling \( (\mathcal{P}')_{R2}^v \) in \( \mathcal{P}' \) with additional cost \( |dO| \).

We will now prove the inequality (2.4) by building a pebbling sequence \( \mathcal{P} \) for \( C \), from an optimal pebbling sequence \( \mathcal{P}' \) for \( C' \), with no additional cost. Let the cost of \( \mathcal{P}' \) be \( Q(C') \). \( \mathcal{P} \), with cost \( Q(C') \), is built from \( \mathcal{P}' \) as follows: (1) Start by copying all the pebblings in \( \mathcal{P}' \) to \( \mathcal{P} \), i.e., \( \mathcal{P} \leftarrow \mathcal{P}' \); (2) for any input vertex \( v \in dI \), the first pebbling of \( v \) with the rule R1 in \( \mathcal{P}' \) is replaced with \( P_{R3NR}^v \) in \( \mathcal{P} \) followed by \( P_{R2}^v \). This produces a valid pebbling sequence \( \mathcal{P} \) for \( C \) with cost \( Q(C') \). □
2.5 Related Work

As discussed earlier, several works have focused on deriving data movement lower bounds for the restricted model that prohibits recomputation. Bilardi et al. [9] defined space complexity under the relaxed model that allows recomputation, and related it to the problem of data movement complexity. Bilardi and Peserico proposed the width decomposition framework in [8] to provide a machine-independent characterization of temporal locality of computations for both recompute and no-recompute based models.

The notion of decomposition has been used previously to obtain tighter lower bounds. Bilardi and Preparata [12] showed, for their specific lower bounding technique based on close-dichotomy-size, that tighter results can be obtained through disjoint-decomposition of DAGs. Our decomposition theorem is more general in the sense that it is independent of any lower bounding technique used, and is applicable for the both the pebble game models that allow or disallow recomputation. Further, we extend the idea to a special case of non-disjoint decomposition in order to derive tighter bounds for certain algorithms.

The idea of tagging/untagging inputs and outputs was utilized by Ballard et al. [4] to obtain non-trivial bounds for various linear algebra algorithms. However, their technique is based on the geometric model based abstraction, that is very different from the CDAG based abstraction that is used in our work.
CHAPTER 3

Lower Bounding Technique Based on Graph Min-cuts

The red-blue pebble game serves as a nice abstraction to model the problem of data movement complexity of algorithms. However, combinatorial nature of the pebble game model poses difficulties in directly using this abstraction for reasoning about tight lower bounds for CDAGs. Hong & Kung hence formulated the problem of finding lower bounds as a graph partitioning problem, called $S$-partitioning, and derived tight lower bounds for several algorithms. Section 3.1.1 provides a summary of Hong & Kung’s $S$-partitioning technique. In this chapter, we propose a novel min-cut based lower bounding technique to bound the data movement complexity of CDAGs. Section 3.3 derives data movement lower bounds for different algorithms using our newly developed technique.

3.1 Background

3.1.1 $S$-partitioning technique

This subsection describes Hong & Kung’s $S$-partitioning technique for the RB pebble game model.
Definition 12 (Dominator set). Given a CDAG $C = (I, V, E, O)$ and a vertex-set $V_s \subseteq V$, a subset $D_s \subseteq V$ is a dominator set for $V_s$, if $D_s$ contains at least one vertex of every path from $I$ to $V_s$.

A vertex-set $V_s$ could have several possible valid dominator sets.

Definition 13 (Minimum set). Given a CDAG $C = (I, V, E, O)$ and a vertex-set $V_s \subseteq V$, minimum set of $V_s$, $M_s$, is the set of vertices in $V_s$ with all its successors outside $V_s$.

Definition 14 (S-partition). Given a CDAG $C = (I, V, E, O)$, an $S$-partition of $C$ is a sequence of $h$ subsets of $V$, $(V_1, \ldots, V_h)$, that satisfies the following properties:

P1 $i \neq j \implies V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{h} V_i = V$.

P2 There is no cyclic dependence between the subsets.

P3 For every subset $V_i, 1 \leq i \leq h$, there exists a dominator set $D_i$ such that $|D_i| \leq S$.

P4 For every subset $V_i, 1 \leq i \leq h$, size of its minimum set, $|M_i| \leq S$.

Hong & Kung showed that corresponding to any pebbling sequence $P$, obtained by playing the RB pebble game with $S$ red pebbles, a $2S$-partition can be constructed with a tight relationship between the size of the $2S$-partition and the cost of $P$.

Theorem 4 ([27]). Any pebbling sequence $P$ obtained by playing the RB pebble game on a CDAG $C$ using at most $S$ red pebbles is associated with a $2S$-partition $\mathcal{H}$ of $C$ such that $S \times h \geq q(P, S) \geq S \times (h - 1)$, where $q(P, S)$ is the cost of $P$ and $h = |\mathcal{H}|$.

Lemma 2 ([27]). Let $H(2S)$ be the minimum number of vertex sets of any $2S$-partition of a CDAG $C$. Then the data movement complexity of $C$ with recomputation, $Q \geq S \times (H(2S) - 1)$. 

26
This key lemma was used by Hong & Kung in [27] to derive tight lower bounds for several algorithms.

### 3.1.2 Definitions from graph theory

In this subsection, we summarize a list of relevant terminologies and definitions from graph theory.

**Definition 15 (Cut).** Given a DAG $G = (V, E)$, a cut, $X = (X, Y)$, of $G$ is defined as any partition of $V$ into two disjoint subsets $X$ and $Y = V \setminus X$ such that $X$ and $Y$ are not empty sets.

**Definition 16 ($s$-$t$ cut).** Given a DAG $G = (V, E)$ and two vertices $s, t \in V$, an $s$-$t$ cut is a cut $X = (X, Y)$ of $G$ such that $s \in X$ and $t \in Y$.

**Definition 17 (Cut-set).** Given a cut $X = (X, Y)$ of a DAG $G = (V, E)$, edge-cut-set of $X$ is the set $E_X = \{(u, v) | u \in X \land v \in Y\} \subseteq E$; and vertex-cut-set of $X$ is the set $V_X = \{u | \exists v[(u, v) \in E_X]\} \subseteq X$. Any vertex $v \in V_X$ is referred to as a cut-vertex.

**Definition 18 (Vertex-cut-size).** Vertex-cut-size of a cut of an unweighted DAG is defined as the cardinality of its vertex-cut-set.

**Definition 19 (Min-cut).** Given an unweighted DAG $G = (V, E)$, a vertex-min-cut of $G$ is a cut (or an $s$-$t$ cut) with minimum vertex-cut-size. Given an unweighted DAG $G = (V, E)$, a edge-min-cut of $G$ is a cut (or an $s$-$t$ cut) with minimum edge-cut-size.
3.2 Min-cut Based Lower Bounding Technique

Our approach on using min-cuts for determining lower bounds was motivated from the observation that Hong & Kung’s 2S-partitioning approach doesn’t account for the internal structure of a CDAG, but essentially focuses only on the boundaries of the vertex sets of a partition. In contrast, our min-cut based approach captures internal space requirements using the abstraction of wavefronts. This is illustrated in the following example: Consider the diamond-DAG $C = (\{v_1\}, V, E, \{v_2\})$ in Figure 3.1. For any value of $S \geq 1$, the set, $V \setminus \{v_1\}$, of all vertices except the one input vertex, forms a valid vertex set of a $2S_{NR}$-partition of $C$, providing a $2S_{NR}$-partition of size $H(2S) = 1$, hence, leading to a trivial lower bound of zero. In general, if any CDAG contains very few inputs and outputs, (specifically, less than $2S$,) a trivial $2S$-partition with just a single vertex set can be formed leading to a lower bound of zero.

Figure 3.1: Diamond DAG. Vertices shaded in black and grey represent input and output vertices, respectively.
We describe our approach below and derive a tight bound for diamond-DAG in Section 3.3.

**Definition 20** (Convex cut). Given a DAG $G = (V, E)$ and a vertex $v \in V$, a convex cut of $G$ w.r.t. $v$ is a cut $X_v = (X_v, Y_v)$ with the following properties:

1. $X_v \cup Y_v = V$ and $X_v \cap Y_v = \emptyset$.
2. $\{v\} \cup \text{Anc}(v) \subseteq X_v$.
3. $\text{Desc}(v) \subseteq Y_v$.
4. $E \cap (Y_v \times X_v) = \emptyset$.

**Definition 21** (Convex vertex-min-cut). Given an unweighted DAG $G = (V, E)$ and a vertex $v \in V$, a convex vertex-min-cut of $G$ w.r.t. $v$, $\overline{X}_v(G)$, is a convex cut of $G$ with minimum vertex-cut-size.

**Definition 22** (Convex edge-min-cut). Given an unweighted DAG $G = (V, E)$ and a vertex $v \in V$, a convex edge-min-cut of $G$ w.r.t. $v$ is a convex cut of $G$ with minimum edge-cut-size.

Consider a pebbling sequence $\mathcal{P}$ for a CDAG $C = (I, V, E, O)$ under the $\text{RB}_{NR}$ model. Consider the time-stamp $t_x$ just after firing of a vertex $x \in V$ in $\mathcal{P}$. Let $X$ denote the set of vertices that were fired before the time $t_x$ in $\mathcal{P}$. (Note that $x \in X$.) If any vertex $v \in X$ has at least one of its successors $s$ not yet fired at time $t_x$, then $v$ is necessarily needed at a later time when $s$ is fired. Since $\text{RB}_{NR}$ game allows a vertex to be fired only once, the computation $v$, that was computed before time $t_x$, has to be “saved” for later use (for computing $s$). This set of “alive” nodes, which
were computed before \( t_x \) and will be needed later, gives rise to a wavefront. We call this wavefront as **schedule wavefront**, which is formally defined below.

**Definition 23** (Schedule wavefront). Given a pebbling sequence \( \mathcal{P} \) for a CDAG \( C = (I, V, E, O) \) and a vertex \( x \in V \), a schedule wavefront of \( \mathcal{P} \) w.r.t. \( x \), \( W_x(\mathcal{P}) \), is the vertex \( x \) and the set of vertices that were fired before \( x \) in \( \mathcal{P} \) with at least one of their successors fired after \( x \), i.e., if \( t_x \) represents the time-stamp at which a vertex \( x \) is fired in \( \mathcal{P} \), then

\[
W_x(\mathcal{P}) = \{ u | t_u \leq t_x \land \exists v[(u, v) \in E \land t_v \geq t_x] \}
\]

Size of a schedule wavefront, \( |W_x(\mathcal{P})| \), essentially signifies the space requirement for the program just after the operation \( x \) has been executed. Note the following close association of a schedule wavefront in a pebbling sequence with the vertex-cut-set of a convex cut in a DAG:

**Proposition 1.** Let \( C = (I, V, E, O) \) be a CDAG with its DAG \( G = (V, E) \). Let \( \mathcal{P} \) be a pebbling sequence for \( C \). If \( X_x \subseteq V \) is the set of vertices that were fired at or before a vertex \( x \in V \) and \( Y_x = V \setminus X_x \) is the set of vertices that are fired after \( x \) in \( \mathcal{P} \), then the partition \( X_x = (X_x, Y_x) \) forms a valid convex cut w.r.t. \( x \) for \( G \) whose vertex-cut-set is the set \( W_x(\mathcal{P}) \).

Let \( W(G, x) \) denote the vertex-cut-set of the convex min-cut \( X_x \). Since a convex vertex-min-cut is a cut with minimum vertex-cut-set size over all possible convex cuts of \( G = (V, E) \) w.r.t. \( x \), we have \( |W(G, x)| \leq |W_x(\mathcal{P})| \). If \( W(G) = \max_{v \in V} |W(G, v)| \) denotes the minimum vertex-cut-size maximized over all vertices in \( G \), then the following lemma provides a relationship between the convex vertex-min-cut and the data movement complexity of a CDAG.
Lemma 3. Given a CDAG $C = (\emptyset, V, E, O)$ with DAG $G = (V, E)$ and no input vertices, for any vertex $x \in V$, $Q \geq 2 \times (|W(G, x)| - S)$. In particular, $Q \geq 2 \times (W(G) - S)$, where, $S$ is the number of red pebbles.

Proof. Consider an optimal pebbling sequence $P$ for $C$ of cost $Q$. Let the schedule wavefront induced by the vertex $x$ in $P$ be $W_x(P)$. Consider the timestep $t_x$ when $x$ is fired in $P$. Since every vertex in $W_x(P)$ has a successor that has not been fired yet, it must be having either a red or a blue pebble on it at the time $t_x$. Since we have only $S$ red pebbles, at least $|W_x(P)| - S$ of them must be having a blue pebble on them. Let $V_B$ denote the set of these vertices with blue pebbles. Each vertex in $V_B$ will have to be red-pebbled with the rule R1 at some point in the future and will incur at least $|W_x(P)| - S$ loads after the time $t_x$. In addition, as $C$ has no input vertices, vertices in $V_B$ belong to the operation set of $C$. A red pebble would have been placed on the vertices in $V_B$ when they were fired in $P$ and later blue-pebbled using the rule R2 leading to at least $|W_x(P)| - S$ stores before timestep $t_x$. Hence, $Q \geq 2 \times (|W_x(P)| - S)$ Thus, we have,

$$Q \geq 2 \times (|W_x(P)| - S) \geq 2 \times (|W(G, x)| - S). \quad \square$$

The above lemma bounds the schedule wavefront size, and thus the space requirement, at a single time-stamp during an execution. It is of interest to capture the space requirements at multiple time-stamps so as to obtain tighter lower bounds. The following theorem makes use of the decomposition theorem to bound the size of the schedule wavefront at multiple locations.

Theorem 5. Given a CDAG $C = (I, V, E, O)$ and some set $dI \subseteq I$ and $dO \subseteq O$, let $I' = I \setminus dI$, $O' = O \setminus dO$, $V' = V \setminus (dI \cup dO)$, $E' = E \cap V' \times V'$ and $C' =$
(I', V', E', O'). Let \{V'_1, V'_2, \ldots, V'_n\} be an arbitrary (not necessarily acyclic) pairwise-disjoint partitioning of V' (i.e., i \neq j \implies V'_i \cap V'_j = \emptyset and \bigcup_{i=1}^{n} V'_i = V') and C'_1, C'_2, \ldots, C'_n be the induced sub-CDAGs of C' (i.e., I'_i = I' \cap V'_i, E'_i = E' \cap V'_i \times V'_i, O'_i = O' \cap V'_i). Then the data movement complexity of C, Q(C), satisfies

\[ Q(C) \geq \sum_{i=1}^{n} 2 \times (W(G'_i) - S) + |dI| + |dO| \]

where, \( G'_i \) represents the DAG of the CDAG \( C'_i \), for \( i = 1, \ldots, n \).

**Proof.** From Corollary 1, we have,

\[ Q(C) \geq Q(C') + |dI| + |dO|. \]  \hspace{1cm} (3.1)

From Theorem 1, we have,

\[ Q(C') \geq \sum_{i=1}^{n} Q(C'_i). \]  \hspace{1cm} (3.2)

From Lemma 3,

\[ Q(C'_i) \geq 2 \times (W(G'_i) - S). \]  \hspace{1cm} (3.3)

Hence from inequalities (3.1), (3.2) and (3.3),

\[ Q(C) \geq \sum_{i=1}^{n} 2 \times (W(G'_i) - S) + |dI| + |dO|. \]  \hspace{1cm} \Box

### 3.3 Analytical Lower Bounds Using Min-cut Approach

In this section, we provide lower bounds for a few algorithms derived using our min-cut based approach.

#### 3.3.1 Diamond DAG

This subsection derives a tight lower bound for Diamond DAG using our min-cut based lower bounding technique.
**Theorem 6** (Lower bound for data movement complexity of diamond DAG). *Given CDAG $C = (I, V, E, O)$ for an $n \times n$ diamond DAG, the data movement complexity of $C$, $Q$, satisfies $Q \geq \frac{(n-2S)^2}{S} = \Omega\left(\frac{n^2}{S}\right)$, where, $S$ is the number of red pebbles.*

*Proof.* Consider an optimal pebbling sequence $\mathcal{P}$ using RB$_{NR}$ pebble game for $C$ with cost $Q$. Consider a vertex $x \in V$ at location $(r, c)$ (i.e., in row-$r$ and column-$c$). Consider a (rotated) bow-tie shaped induced sub-DAG $G_x = (V_x, E_x)$ around $x$, as shown in Figure 3.2, that consists of vertices that lie inside the two $m \times m$ squares, whose corners are at $[(r, c - m + 1), (r + m - 1, c)]$ and $[(r - m + 1, c), (r, c + m - 1)]$. Let $C_x$ be the induced sub-CDAG of $V_x$ in $C$.

![Figure 3.2: Diamond DAG with bow-tie shaped sub-DAG](image)

There are $2m$ vertex-disjoint paths from the ancestor set of $x$ to its descendant set. Any convex cut of $G_x$ w.r.t. $x$ should have at least one vertex from each of these...
2m disjoint paths in its vertex-cut-set. Hence, vertex-cut-size of a convex vertex-min-
cut of $G_x$ w.r.t. $x$, $|\mathcal{W}(G_x, x)| \geq 2m$. From Lemma 3, we have, the data movement
complexity of sub-CDAG $C_x$, $Q(C_x) \geq 2 \times (2m - S)$.

A rectangle of size $n \times 2m$ can be partitioned into $\lfloor (n - m)/m \rfloor$ disjoint bow-ties.
An $n \times n$ diamond DAG can itself be partitioned into $\lfloor n/2m \rfloor$ disjoint rectangles of size $n \times 2m$. Hence, an $n \times n$ diamond DAG can be partitioned into $\lfloor (n - m)/m \rfloor \times \lfloor n/2m \rfloor$
disjoint bow-tie shaped sub-CDAGs. Hence, using Theorem 5, we get

$$Q \geq 2 \times (2m - S) \times \left\lfloor \frac{n - m}{m} \right\rfloor \times \left\lfloor \frac{n}{2m} \right\rfloor.$$

By setting $m = S$, we get:

$$Q \geq 2S \times \left\lfloor \frac{n - S}{S} \right\rfloor \times \left\lfloor \frac{n}{2S} \right\rfloor \quad \geq \quad 2S \times \frac{n - S - S + 1}{S} \times \frac{n - 2S + 1}{2S} \quad \geq \quad \frac{(n - 2S)^2}{S}.$$

This bound is tight by a factor of 2 as the following theorem states.

**Theorem 7** (Upper bound for data movement complexity of diamond DAG). There
exists an implementation for the $n \times n$ diamond DAG with a data movement cost of
$2n^2/S$.

**Proof.** To prove the upper bound, we consider the following pebbling implementation,
where $Rx(i, j)$ represents application of rule $Rx$ on the vertex at location $(i, j)$.

```plaintext
for b = 1 : n step m
  for j = 1 : n
    for i = b : min(b + m - 1, n)
      if (i = b and i \neq 1): R1(i - 1, j)
      R3_{NR}(i, j)
      if (i = b and i \neq 1): R4(i - 1, j)
      if (j \neq 1): R4(i, j - 1)
      if (i = b + m - 1 and i \neq n): R2(i, j)
```

34
This implementation uses exactly $S = m + 2$ red pebbles. For each of the upper $\lceil n/m \rceil - 1$ bands of width $m$, it performs $n$ loads; for each of the lower $\lceil n/m \rceil - 1$ bands of width $m$, it performs $n$ stores. This gives an overall data movement cost of $2n \left( \lceil \frac{n}{S-2} \rceil - 1 \right)$. □

### 3.3.2 Fast Fourier transform (FFT)

In this subsection, we derive an tight lower bound, that matches the upper bound for FFT.

**Theorem 8** (Lower bound for data movement complexity of FFT). *For the $n$-point FFT, its data movement complexity $Q$ satisfies*

$$Q \geq \frac{2n \log(n)}{\log(S)} \times (1 - \epsilon),$$

*where, $S$ is the number of red pebbles, and $\epsilon$ tends to 0 for large values of $n$, $S$, and $n/S$.*

**Proof.** Consider the CDAG $C_m$ for an $m$-point FFT, with the input vertices removed. Let $G_m = (V_m, E_m)$ be the DAG of $C_m$. $V_m$ contains a set of $m$ predecessor-free vertices $A$ and $m$ successor-free vertices $B$. A DAG of an FFT computation has the following properties: There is a non-empty vertex-disjoint path from each predecessor-free vertex to a distinct successor-free vertex. Also, each successor-free vertex is dependent on (i.e, has a non-empty path from) all the predecessor-free vertices.

Consider any optimal pebbling sequence $P_m$ for $C_m$ of cost $Q_m$. Let $t_o$ be the time-stamp at which the first successor-free vertex $o \in B$ is fired in $P_m$. Let $X$ be the set of vertices fired strictly before $t_o$, and $Y = V_m \setminus X$ be the set of vertices fired during
or after $t_o$. Since each successor-free vertex is dependent on the $m$ predecessor-free vertices, $A \subseteq X$. By construction, $B \subseteq Y$. Since $G_m$ contains $m$ vertex-disjoint paths from $A$ to $B$, size of the corresponding schedule wavefront, $|W_o(P)| \geq m$. Hence, $m$ is the lower bound on the size of schedule wavefront at the time of firing the first successor-free vertex in any pebbling sequence for $C_m$. Hence, $Q_m \geq 2 \times (m - S)$.

A CDAG for the $n$-point FFT (of height $\log(n)$ with input vertices removed) can be decomposed into $\lfloor n/m \rfloor \times \lfloor \log(n)/\log(m) \rfloor$ disjoint sub-CDAGs, each equivalent to $C_m$. By setting $m = S \log(S)$, we get:

$$Q \geq \left\lfloor \frac{n}{S \log(S)} \right\rfloor \times \left\lfloor \frac{\log(n)}{\log(S \log(S))} \right\rfloor \times 2(S \log(S) - S)$$

$$\geq \left( \frac{n}{S \log(S)} - 1 \right) \times \left( \frac{\log(n)}{\log(S \log(S))} - 1 \right) \times 2(S \log(S) - S)$$

$$\geq \frac{2n \log(n)}{\log(S)} \times (1 - \epsilon). \quad \square$$

This bound is tight as a well-known tiled version of FFT consists of decomposing it into $\log(n)/\log(S)$ stages of $n/S$ sub-FFTs of size $S$. For each sub-FFT the $S$ inputs are loaded, the computation is spill-free, and the $S$ output are stored, leading to a data movement cost of $2n \log(n)/\log(S)$.

### 3.3.3 9-point Jacobi 2D (J2D)

We consider the Jacobi computation that computes, at time step $t$, the value of every point of an $n \times n$ array $A$ in terms of its nine neighbors computed at time step $t - 1$ as follow:

$$A[i][j] = (B[i-1][j-1] + B[i-1][j] + B[i-1][j+1]$$
$$\quad \quad + B[i][j-1]$$
$$\quad + B[i][j] + B[i][j+1] + B[i+1][j-1]$$
$$\quad + B[i+1][j] + B[i+1][j+1]) \times \text{cnst};$$

**Theorem 9** (Lower bound for data movement complexity of J2D). *For the 9-point Jacobi computation of size $n \times n$ with $T$ time steps, its data movement complexity,
\( Q \), satisfies

\[
Q \geq \frac{0.75n^2T}{\sqrt{S}}
\]

where \( S \) is the number of red pebbles.

Proof. Jacobi 2D has a three-dimensional iteration space of size \( n^2 \times T \) without accounting for the inputs from timestep \( T_0 \). Consider a partition of the iteration space (and the corresponding CDAG) into cubes (i.e., into sub-CDAGs) of size \( m^3 \). We have \( \lfloor n/m \rfloor^2 \times \lfloor T/m \rfloor \) such cubes. For each cube, let \( x \) be the vertex at the center of the cube. The ancestor set of \( x \) contains the lower face of the cube. The descendant set of \( x \) contains the upper face. We have \( m^2 \) disjoint vertical paths connecting each point in the lower face with a distinct point in the upper face of the cube. Vertex-cut-set, \( \text{W}(G_x, x) \) (where \( G_x \) is the DAG of the current sub-CDAG), of any convex cut w.r.t. \( x \) must have at least one vertex from each of those \( m^2 \) disjoint paths. Hence, \( |\text{W}(G_x, x)| \geq m^2 \). Hence, from Lemma 3, the data movement complexity of each sub-CDAG is \( 2(m^2 - S) \).

Considering all \( \lfloor n/m \rfloor^2 \times \lfloor T/m \rfloor \) sub-CDAGs, using Theorem 5, we get the data movement complexity of the full CDAG,

\[
Q \geq \left\lfloor \frac{n}{m} \right\rfloor^2 \times \left\lfloor \frac{T}{m} \right\rfloor \times 2(m^2 - S) + |I| + |O|.
\]
We choose an $m$ that maximizes the right-hand side of the above inequality. By setting $m = \sqrt{3S}$,

$$Q \geq \left( \frac{n}{\sqrt{3S}} \right)^2 \times \left( \frac{T}{\sqrt{3S}} \right) \times 2 \left( \left( \sqrt{3S} \right)^2 - S \right) + n^2 + n^2$$

$$\geq \left( \frac{n}{\sqrt{3S}} \right)^2 \times \left( \frac{T}{\sqrt{3S}} \right) \times 4S \quad = \quad \frac{4n^2T}{3\sqrt{3S}} \quad \geq \quad \frac{0.75n^2T}{\sqrt{S}}. \quad \Box$$

**Theorem 10** (Upper bound for data movement complexity of J2D). *There exists an implementation for 9 point Jacobi computation of size $n \times n$ with $T$ time steps with a data movement cost of

$$\frac{8n^2T \times (1 + \epsilon)}{\sqrt{S}} + 2n^2,$$

where $\epsilon$ tends to zero for $n \gg \sqrt{S}$.\)

**Proof.** Jacobi 2D computation contains a three-dimensional iteration space of size $n^2 \times T$, with dependencies along directions $(1, 0, 0)^T$, $(1, -1, 0)^T$, $(1, 1, 0)^T$, $(1, 0, -1)^T$, $(1, 0, 1)^T$, $(1, -1, -1)^T$, $(1, -1, 1)^T$, $(1, 1, -1)^T$ and $(1, 1, 1)^T$. Consider a skewed version of this iteration space such that dependencies are along directions $(1, 1, 1)^T$, $(1, 0, 1)^T$, $(1, 2, 1)^T$, $(1, 1, 0)^T$, $(1, 1, 2)^T$, $(1, 0, 0)^T$, $(1, 0, 2)^T$, $(1, 2, 0)^T$ and $(1, 2, 2)^T$. This skewed iteration space can be partitioned into vertical 3D columns each of height at most $T$ and cross-section $\sqrt{S} \times \sqrt{S}$. (Columns near the boundaries are shorter and progressively get taller until it reaches a height of $T$ as we move towards the center.) There are $\left( (N + T)/\sqrt{S} \right)^2$ such columns. Since the dependencies are only along forward directions in this skewed iteration space, starting from the first column at corner
(0, 0, 0)^T$, each column can be computed consecutively in lexicographic order. We refer to each column using its index $(i, j)$. Each column needs to load two planes of data computed by column $(i - 1, j)$ and two planes of data computed by column $(i, j - 1)$. Similarly, each column needs to store four planes of data whose sizes are same as that of loads. Size of the plane loaded from column $(i - 1, j)$ is $\min(T\sqrt{S}, i\sqrt{S}, n - i\sqrt{S})$ and size of the plane loaded from column $(i, j - 1)$ is $\min(T\sqrt{S}, j\sqrt{S}, n - j\sqrt{S})$. (Boundary element reads are also captured here.) Due to symmetry of the skewed space, it can be shown that this leads to total loads and stores of $8 \left\lceil \frac{n}{\sqrt{S}} \right\rceil^2 T\sqrt{S}$. There are $n^2$ inputs and outputs which need to be loaded/stored just once. Hence, the data movement complexity $Q$ for J2D satisfies

$$Q \leq 8 \times \left\lceil \frac{n}{\sqrt{S}} \right\rceil^2 T\sqrt{S} + 2n^2$$

$$\leq 8 \times \left( \frac{n}{\sqrt{S}} + 1 \right)^2 T\sqrt{S} + 2n^2$$

$$\leq \frac{8n^2T \times (1 + \epsilon)}{\sqrt{S}} + 2n^2,$$

where $\epsilon$ tends to zero for $n \gg \sqrt{S}$. □

### 3.3.4 Conjugate Gradient (CG)

Many compute intensive applications involve the numerical solution of partial differential equations (PDEs), e.g., solving a heat equation on a two-dimensional plate. The first step in numerical solution of such problems involves discretization, where the continuous domain is reduced to discrete points at regular intervals. The problem is then solved only at these discrete points, called computational grid. Solution to such discretized problems involves solving a system of linear equations at each
timestep till convergence. These linear systems, of the form $Ax = b$, are typically made of banded sparse matrix $A$ with repetitive pattern. Hence, in practice, the elements of $A$ are not explicitly stored. Instead, their values are directly embedded in the program as constants thus eliminating the space requirement and the associated I/O cost for the matrix.

The conjugate gradient method [26] is one of several popular methods to iteratively solve such linear system. CG maintains 3 vectors at each timestep - the approximate solution $x$, its residual $r = Ax - b$, and a search direction $p$. At each step, $x$ is improved by searching for a better solution in the direction $p$. Each iteration of CG involves one sparse matrix-vector product, three vector updates, and three vector dot-products. The complete pseudo-code is shown in Algorithm 3.1.

```
1  Function CG(x0)
2      Input : Initial guess $x_0$
3      Output: $x$
4  x ← x0 ;
5  p ← r ← b − Ax;
6  repeat
7      v ← Ap;
8      b ← (r.r); // Dot product
9      a ← b/ (p.v); // Dot product
10     x ← x + ap; // AXPY
11     r ← r − av; // AXPY
12     g ← (r.r) / b; // Dot product
13     p ← r + gp; // AXPY
14   until (r.r) is small;
15  return x;
```

**Algorithm 3.1:** Conjugate Gradient method.
Theorem 11 (Lower bound for data movement complexity of CG). For a $d$-dimensional grid of size $n^d$, the minimum data movement cost to solve the linear system using CG with $S$ red pebbles, $Q(S)$, satisfies $Q(S) \geq 6n^dT$, when $n^d \gg S$; where, $T$ represents the number of outer loop iterations.

Proof. Let $C$ and $G = (V, E)$ denote the CDAG and its DAG, respectively, of CG with the inputs removed. We will use non-disjoint decomposition (refer Definition 11) to partition $C$ into $2T+1$ sub-CDAGs. Let $v(s, t)$ denote the vertex in $V$ corresponding to computation of a scalar value $s$ at timestep $t$ in Algorithm 3.1. For example, $v(a, 1)$ denotes computation of the scalar $a$ at Line 7 during the first timestep.

Consider the subset $V' = \{v(a, 1), v(g, 1), v(a, 2), v(g, 2), \ldots, v(a, T), v(g, T)\} \subset V$. Let $\{C'(a, 1), C'(g, 1), C'(a, 2), C'(g, 2), \ldots, C'(a, T), C'(g, T), C''\}$ be the partition obtained through nondisjoint decomposition of $C$ w.r.t. $V'$. Note that $n^d$ vertices corresponding to the computation of $n^d$ elements of the vector $r$ at Line 9 are shared between sub-CDAGs $C'(a, t)$ and $C'(g, t)$, for $t = 1, \ldots, T$. Also $n^d$ vertices corresponding to the computation of the vector $p$ at Line 11 are shared between sub-CDAGs $C'(g, t)$ and $C'(a, t+1)$, for $t = 1, \ldots, T-1$.

Consider any sub-CDAG $C'(a, t)$, for $t = 1, \ldots, T$, and the vertex $x = v(a, t)$ in $C'(a, t)$. The $2n^d$ predecessor vertices of $x$, corresponding to elements of vectors $p$ and $v$, have vertex-disjoint paths to $\text{Desc}(x)$ (to the vertices corresponding to computations in Lines 8 and 9). This gives us a convex vertex-min-cut of vertex-cut-size $|\underline{W}(C'(a, t), x)| \geq 2n^d$. Hence, data movement complexity of the sub-CDAG $C'(a, t)$ with $S + 1$ red pebbles satisfies $Q(C'(a, t), S + 1) \geq 2 \times (2n^d - S - 1)$.

Similarly, considering any sub-CDAG $C'(g, t)$, for $t = 1, \ldots, T$, and the vertex $y = v(g, t)$ in $C'(g, t)$, we obtain a convex vertex-min-cut with vertex-cut-size
\[ |W(G'(g,t),y)| \geq n^d, \] due to the vertex-disjoint paths from the predecessors corresponding to computation of \( r \) to \( \text{Desc}(y) \) (to the vertices corresponding to computation of \( p \) at Line 11). Hence, 
\[ Q(C'(g,t), S + 1) \geq 2 \times (n^d - S - 1). \]

The original CDAG has \( n^d \) inputs and \( n^d \) outputs. By applying Theorem 2 on the sub-CDAGs, we obtain:
\[
Q(S) \geq T \times (2(2n^d - S - 1)) + T \times (2(n^d - S - 1)) + n^d + n^d
= 2T \times (3n^d - 2S - 2) + 2n^d
\]
which tends to \( 6n^dT \) when \( n^d \gg S \). \( \square \)

**Theorem 12** (Upper bound for data movement complexity of CG). *Algorithm 3.1 incurs a data movement cost of \( 12n^dT + 5n^d \).*

*Proof.* Lines 2 and 3 incur a cost of \( 5n^d \) to load/store vectors \( x_0, x, b, r \) and \( p \). Line 6 incurs \( n^d \) data movements per time-step to load \( r \). SpMV operation at Line 5 is equivalent to a single time-step of stencil computation, which can be computed using a spatial tiled implementation with data movement cost of \( n^d \) (to load \( p \)) + \( n^d \) (to store \( v \)) + \( n^d(d - 1)/S^{1/(d - 1)} \) (for inter-tile spills), which tends to \( 2n^d \) for large values of \( S \). Hence, Line 5 requires \( 2n^d \) data movements per time-step. In addition, computation of Line 7 can be interleaved with Line 5, providing a cost of \( 2n^d \) to compute both lines 5 and 7. Line 8 requires \( 3n^d \) data movements. Lines 9 and 10 (and Line 12) can be interleaved providing a cost of \( 3n^d \) per time-step. Finally, Line 11 requires \( 3n^d \) data movements. So, in total, Algorithm 3.1 has a data movement cost of \( 12n^dT + 5n^d \). \( \square \)
### 3.3.5 Summary

Table 3.1 summarizes lower and upper bound results derived in this section. Compared to Hong & Kung’s $S$-partitioning technique, our method is simpler to apply since in case of $S$-partitioning technique, careful reasoning over all possible $S$-partitions has to be made to obtain valid tighter lower bounds. In contrast, in our min-cut based technique, it is sufficient to consider min-cuts with respect to any vertex to obtain a valid bound, and carefully choosing a few vertices would provide a tighter bound. We noted earlier that Hong & Kung’s original $S$-partitioning technique would only provide trivial lower bounds for CDAGs with small input and output set. We would also like to note that our min-cut based technique developed in this chapter does not perform well on CDAGs that have high fan-out input vertices and very little reuse of vertices in the operation set. This includes matrix-matrix multiplication and many other numerical linear algebra applications. Hong & Kung’s $S$-partitioning

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Lower bound</th>
<th>Upper bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>Diamond DAG</td>
<td>$\frac{n^2}{S}$</td>
<td>$\frac{2n^2}{S}$</td>
</tr>
<tr>
<td>FFT</td>
<td>$\frac{2n \log(n)}{\log(S)}$</td>
<td>$\frac{2n \log(n)}{\log(S)}$</td>
</tr>
<tr>
<td>9-point Jacobi 2D</td>
<td>$\frac{0.75n^2T}{\sqrt{S}}$</td>
<td>$\frac{8n^2T}{\sqrt{S}}$</td>
</tr>
<tr>
<td>Conjugate gradient</td>
<td>$6n^dT$</td>
<td>$12n^dT$</td>
</tr>
</tbody>
</table>

Table 3.1: Min-cut based lower bounds and corresponding upper bounds for different algorithms. (Lower order terms are ignored in the summary.)
technique and geometric reasoning based techniques [29, 4, 15, 19] are well suited for these applications.

3.4 Related Work

Hong & Kung provided the first characterization of the data movement complexity problem using the red-blue pebble game and 2S-partitioning of CDAGs [27]. Several works followed Hong & Kung’s work on data movement complexity in deriving lower bounds on data accesses. Aggarwal et al. [2] provided several lower bounds for sorting algorithms. Savage [45, 46] presented a version of Hong & Kung’s results using the notion of S-span to simplify the proofs and extend it to hierarchical memories. This model has been used in several works [43, 48]. Irony et al. [29] provided a new geometric reasoning based proof of the Hong & Kung result on data movement complexity of matrix multiplication and developed lower bounds on communication for sequential and parallel matrix multiplication. Demmel et al. [4, 3, 17, 51, 15, 33] generalized the technique in [29] and developed lower bounds as well as optimal algorithms for several linear algebra computations including QR and LU decomposition and all-pairs shortest paths problem. Ranjan et al. [42] develop tighter bounds for binomial graph and FFT using a new technique called boundary flow technique that is related to the vertex isoperimetric parameter of graphs. Bilardi et al. [9, 8] developed the notion of space complexity and related it to data access complexity.

Bilardi and Preparata [12] developed the notion of the closed-dichotomy size, $\beta(w)$ of a DAG $G$ that is used to provide a lower bound on the data movement complexity in those cases where recomputation is not allowed. The closed-dichotomy technique can be summarized as follows: A subset of vertices $W \subseteq V$ in a DAG $G = (V, E)$ is called
a closed set if whenever a vertex \( v \in W \) then its predecessor \( u \) also belongs to \( W \). \( B_{out}(W) \) is defined as the set of vertices in \( W \) with at least one successor in \( V \setminus W \). Finally, the closed-dichotomy size, \( \beta(w) = \min\{|B_{out}(W)| : W \subseteq V, W \text{ is closed}, |W| = w\} \). This notion is very closely related to our schedule wavefronts and graph min-cuts. 

For a given pebbling sequence \( \mathcal{P} \), we define a schedule wavefront w.r.t. a vertex \( x \) as the set of vertices that were fired before \( x \) and have at least one successor fired after \( x \). In [12], the set of vertices \( W \) fired before \( x \) form a closed set and \( B_{out}(W) \) can be considered to be equivalent to our schedule wavefront. (Note that \( B_{out}(W) \) is not defined w.r.t. a particular vertex, but w.r.t. a closed set \( W \).) The two techniques have the following key difference: We define our schedule wavefront \( W_x(\mathcal{P}) \) w.r.t. any vertex \( x \) and seek to find a lower bound on \( |W_x(\mathcal{P})| \); while in [12], an integer value \( w \) is fixed and \( B_{out}(W) \) is defined over all possible closed subsets \( W \) of size \( w \) and they seek a minimum over \( |B_{out}(W)| \). The need to find a minimum over all closed sets \( W \) such that \( |W| = w \) makes their formulation combinatorial in nature. In contrast, a close relationship between our schedule wavefront and graph min-cuts allows us to develop a polynomial time heuristics (in the following chapter) to automatically analyze lower bounds of CDAGs.

The purpose of the two techniques are quite different. We derived data movement lower bounds of computations using our technique in a two-level serial machine model and intend to extend it to develop a practical heuristic (in Chapter 4) in order to be able to automatically derive lower bounds of computations. Bilardi and Preparata [12] extended their technique to parallel machine model and successfully employed their technique to analytically derive lower bounds to slowdown of simulation of DAGs on an \( n \)-processor array by a \( p \)-processor array (for \( p \leq n \)).
CHAPTER 4

Automatic Min-cut Based Lower Bounds

In Chapter 3, we developed a min-cut based lower bounding technique, where convex vertex-min-cuts were used to bound the data movement complexity of CDAGs. In this chapter, we intend to develop techniques to automate the process of finding (non-parametric) lower bounds for any arbitrary CDAG.

In Section 4.1, we develop a method to automatically compute convex vertex-min-cut of DAGs using standard max-flow algorithm. Section 4.2 details techniques to automatically extract CDAG from any input program, and presents a heuristic for automatic analysis of arbitrary CDAGs. Effectiveness and uses of our automated approach are shown in Section 4.3.

4.1 Algorithm for Convex Vertex-Min-Cut

The problem of finding a convex vertex-min-cut in a DAG can be represented as an integer programming problem with the property that its constraint matrix is totally-unimodular. This allows us to solve the problem in polynomial time using linear programming (LP). Let \( C = (I, V, E, O) \) be a CDAG. Then then finding a
convex vertex-min-cut in the DAG $G = (V, E)$ can be formulated as follows:

Minimize $\sum_{v \in V} c_v$

subject to

$\forall (v, w) \in E$, \hspace{1em} \hspace{1em} $t_w \geq t_v$

$\forall (v, w) \in E$, \hspace{1em} \hspace{1em} $c_v \geq t_w - t_v$

$t_x = 0$

$\forall v \in \text{Anc}(x)$, \hspace{1em} $t_v = 0$

$\forall v \in \text{Desc}(x)$, \hspace{1em} $t_v = 1$

$\forall v \in V$, \hspace{1em} $0 \leq t_v, c_v \leq 1$ \hspace{1em} (4.1)

The above formulation minimizes the vertex-cut-size of a convex cut $(X_x, Y_x)$ w.r.t. a vertex $x \in V$. Each vertex $v \in V$ is associated with two non-negative variables $c_v$ and $t_v$. Variable $c_v$ captures the vertex-cut-size, and variable $t_v$ determines if $v$ belongs to the set $X_x$ or $Y_x$, depending on whether $t_v = 0$ or $t_v = 1$, respectively. Convexity is enforced by the first constraint, where, given an edge $(v, w) \in E$, if $v$ is in $Y_x$, then $w$ is also forced to be in $Y_x$. The second constraint $c_v \geq t_w - t_v$ forces a vertex $v \in X_x$ to be accounted as cut-vertex whenever $v$ has at least one successor in $Y_x$. Finally, vertices $v \in \{x\} \cup v \in \text{Anc}(x)$ (resp. $v \in \text{Desc}(x)$) are restricted to the set $X_x$ (resp. $Y_x$) by setting $t_v = 0$ (resp. $t_v = 1$). The minimization objective $\left(\min \sum_{v \in V} c_v\right)$ implicitly ensures that $c_v \leq 1$ and $t_v \leq 1$.

Even though the above LP based formulation of convex vertex-min-cut can be solved in polynomial time, its running time might be long in practice, especially, on CDAGs with millions of vertices. The well known \textit{max-flow min-cut theorem} from optimization theory allows to solve the \textit{standard min-cut} problem (that minimizes the edge-cut-size of an $s$-$t$ cut) as a dual problem of finding maximum amount of flow in a DAG, called max-flow. \textbf{(Note:} We refer to the problem of finding an $s$-$t$ cut in
a graph, that minimizes the edge-cut-size, as the standard min-cut problem.) There are specialized algorithms, such as Ford-Fulkerson method, Push-relabel algorithm, etc., to efficiently solve the max-flow problem. These specialized max-flow algorithms are several times faster in practice compared to their corresponding LP formulations. Hence, it is of interest to somehow utilize these techniques to efficiently solve our related problem of finding convex cuts. In the Subsection 4.1.2, we will show a construction for a modified graph, corresponding to the DAG of a CDAG, that enables us to directly use standard max-flow algorithms to obtain convex vertex-min-cut of the original DAG. In Subsection 4.1.3, we prove the correctness of our method.

4.1.1 Overview

The standard min-cut problem finds an s-t cut with minimum edge-cut-size in a graph. A convex-cut \((X_x, Y_x)\) differs from a standard cut in the following two ways:

\[\text{C1} \quad \text{A convex-cut is formed w.r.t. a vertex } x \text{ s.t. } \text{Anc}(x) \subseteq X_x \text{ and } \text{Desc}(x) \subseteq Y_x.\]

\[\text{C2} \quad \text{There are no edges from vertices in the set } Y_x \text{ to vertices in } X_x.\]

In this subsection, we will show how to use a standard min-cut algorithm to obtain a valid convex-cut with minimum edge-cut-size for a DAG. For our lower bounding purposes, we are interested in finding a convex-cut with minimum vertex-cut-size. However, it is easier to describe the idea for the case of minimizing edge-cut-size. Details for obtaining convex vertex-min-cut using standard min-cut algorithm are presented in the next subsection.

We will first show how to enforce constraint C1 in a standard min-cut problem. We will transform the original graph \(G\) into a new graph \(G'\) as follows: Let \(x\) be the vertex w.r.t. which a convex-cut is required. (1) The vertices \(\{s\} \cup \text{Anc}(x)\) in \(G\) are
contracted into a single source vertex $s'$ in $G'$; (2) The vertices $\{t\} \cup \text{Desc}(x)$ in $G'$ are contracted into a single target vertex $t'$ in $G''$. Now, from the cut $(X', Y')$, returned by a standard min-cut algorithm on $G'$, we obtain the cut $(X, Y)$ for $G$ by replacing $s' \in X'$ with $\{s\} \cup \text{Anc}(x)$ and $t' \in Y'$ with $\{t\} \cup \text{Desc}(x)$. Note that for the case of convex vertex-min-cut, this step is bit more involved, since directly contracting $\{s\} \cup \text{Anc}(x)$ into $s'$ might lead to a convex-cut with non-minimum cut-vertex-size. Details for convex vertex-min-cut are presented in Subsection 4.1.2.

Now, we will show how to enforce constraint C2, i.e., given a graph $G'$, obtained after applying the above transformation to the original graph $G$, we will transform $G'$ into another graph $G''$ such that a standard min-cut on $G''$ provides us a cut $(X', Y')$ for $G'$ with no edge crossing from $Y'$ to $X'$. Recall that the objective of a standard min-cut problem is to find a partition $(X, Y)$ for a graph such that the source $s \in X$ and target $t \in Y$ and the cut-cost, i.e., sum of weights of the edges from $X$ to $Y$, is minimized. Any number of edges (with any weight) are allowed to cross from $Y$ to $X$ and their weights are not accounted for in the cut-cost. However, a convex cut disallows any edge from $Y$ to $X$. The idea is to enforce this constraint by modifying $G' = (V', E')$ to a weighted (non-acyclic) digraph $G'' = (V', E'', W'')$ by adding an edge $(v', u')$ with weight $\infty$ for every edge $(u', v') \in E'$. Hence, from $G' = (V', E')$, a new graph $G'' = (V', E'', W'')$ is created, where $E'' = E' \cup \{(u', v')|(u', v') \in E'\}$ and $W''(e) = 1$ if $e \in E'$, and $W''(e) = \infty$ otherwise. If any cut $(X'', Y'')$ for $G''$ contains a vertex $u' \in Y''$ and $v' \in X''$ such that $(u', v') \in E'$, then we would have the edge $(v', u') \in E''$ that crosses from $X''$ to $Y''$, leading to a cut-cost of $\infty$. It is easy to see that there exists a cut in $G''$ with finite cut-cost. In fact, by removing all the out-edges of the source vertex $s' \in V'$, we would obtain a valid cut $(\{s'\}, V' \setminus \{s'\})$.
with finite cut-cost. Hence, by using a standard min-cut algorithm to find a cut in \( G'' \), it is impossible to have an edge \( e' \in E'' \) as a cut-edge s.t. \( e' \in E' \).

It can be shown that by applying these two transformations to a DAG \( G \), we would obtain a valid convex-cut with minimum edge-cut-size using a standard min-cut algorithm.

The problem of interest for us is to find a convex vertex-min-cut in a graph. There are well known techniques to convert a standard vertex-min-cut problem to a standard edge-min-cut problem. It involves transforming an unweighted graph \( G \), whose (non-convex) vertex-min-cut is to be found, into a new unweighted graph \( G' \), as detailed below, such that a standard edge-min-cut algorithm run on \( G' \) returns a vertex-min-cut for \( G \). \( G' = (V', E') \) is created from \( G = (V, E) \) as follows: (1) each vertex \( v \in V \) is replaced with two vertices \( v_h \) and \( v_t \) and an edge \( (v_h, v_t) \) is created; (2) for each edge \( (u, v) \in E \) an edge \( (u_t, v_h) \) is created in the transformed graph. A transformation similar to this, in addition to the transformations detailed above earlier can be used to convert the problem of finding convex vertex-min-cut to a standard edge-min-cut problem.

In the following subsection, we formalize the method to transform an unweighted DAG \( G \) to a weighted (non-acyclic) digraph \( G' \), in order to find convex vertex-min-cut for \( G' \) using standard edge-min-cut for \( G \).

### 4.1.2 Construction of transformed graph

Let \( G = (V, E) \) be a DAG and \( x \in V \) be the vertex with respect to which a convex-cut is required. We will create a weighted (non-acyclic) digraph \( G_m = (V_m, E_m, W_m) \) corresponding to \( G \). We define dead ancestors of a vertex \( v \) as follows:
**Definition 24** (Dead ancestor set). *Given a DAG $G = (V, E)$, dead ancestor set, $\text{DAnc}(v)$, of a vertex $v \in V$ is the set $\{u|u \in \text{Anc}(v) \land \forall w : (u, w) \in E[w \in \text{Anc}(v)]\}$. 

Informally, the dead ancestor set is the set of ancestors of a vertex whose successors are also within the ancestor set. It can be seen that $\text{DAnc}(x)$ cannot be part of the cut-vertex-set of any convex-cut for $G$ w.r.t. $x$. $\text{DAnc}(x)$ will be contracted to $s$ and $\text{Desc}(x)$ will be contracted to $t$ in $G_m$, and the remaining vertices $v \in V$ will be split into $v_h$ and $v_t$. The process is formally detailed below. Let $V' = V \setminus (\text{DAnc}(x) \cup \text{Desc}(x))$. $G_m$ is created as follows:

1. Add source and target vertices $s$ and $t$ to $V_m$.

2. Corresponding to each vertex $v \in V'$, add a pair of vertices $v_h$ and $v_t$ to $V_m$ and add edge $e = (v_h, v_t)$ of weight one to $E_m$. We call these edges as **type-1** edges. For simplicity, in the remainder of this chapter, any vertex $v$ referenced with a subscript $h$ (resp. $t$) denotes the vertex $v_h \in V_m$ (resp. $v_t \in V_m$) that was created corresponding to $v \in V'$.

3. Create a set of edges $\{(s, v_h)|v \in \text{Anc}(x) \cap V'\}$ of weight one and add them to $E_m$.

4. Corresponding to any edge $e \in \{(u, v)|(u, v) \in E \land u, v \in V'\}$, add an edge $(u_t, v_h)$ of weight one to $E_m$.

5. Corresponding to any edge $e \in \{(u, v)|(u, v) \in E \land u \in V' \land v \in \text{Desc}(x)\}$, add an edge $(u_t, t)$ of weight one to $E_m$. We will call the set of edges created above in steps 3, 4 and 5 as **type-2** edges.
6. For any edge \( e \in \{(u, v) \mid (u, v) \in E \land u, v \in V'\} \), create an edge \((v_h, u_h)\) of weight \( \infty \) and add it to \( E_m \). We will call these edges as \textbf{type-3} edges.

In summary, we have, \( V_m = \{s, t\} \cup \{v_h, v_t\} \mid v \in V'\};

\[
E_m = \{(s, v_h)\mid v \in V'\} \cup \{(v_h, v_t)\mid v \in V'\} \cup \{(u_t, v_h)\mid u, v \in V' \land (u, v) \in E\} \cup \{(v_h, u_h)\mid u, v \in V' \land (u, v) \in E\} \cup \{(v_t)\mid v \in V' \land (v, \text{Desc}(x)) \in E\};
\]

and edge weight function, \( W_m : E_m \to \mathbb{Z} \), where, \( W_m(e) = \infty \) if \( e \) is of type-3, and \( W_m(e) = 1 \) otherwise. The graph \( G_m \) is passed to a standard min-cut algorithm to obtain a convex vertex-min-cut for \( G \). In the following sections, the notation \( W_m(E') \), where \( E' \subseteq E_m \) is a subset of edges, denotes \( W_m(E') = \sum_{e \in E'} W_m(e) \).

### 4.1.3 Correctness

In this subsection, we will prove the correctness of our method.

**Theorem 13.** Given a DAG \( G = (V, E) \) and a vertex \( x \in V \), let \( G_m = (V_m, E_m, W_m) \) be the transformed digraph constructed w.r.t. \( x \) as shown in Subsection 4.1.2. Let \( E_X \) be the edge-cut-set of a standard min-cut \( X = (X, Y) \) of \( G_m \), and let \( V_{X_x} \) be the vertex-cut-set of a convex vertex-min-cut \( X_x = (X_x, Y_x) \) of \( G \) w.r.t. any vertex \( x \in V \). Then, \( W_m(E_X) = |V_{X_x}| \).

We will first prove that \( W_m(E_X) \leq |V_{X_x}| \) by showing that corresponding to \( X_x = (X_x, Y_x) \), there exists a cut \( X = (X', Y') \) for \( G_m \) with edge-cut-set \( E_X \) consisting of edges of weight one and of size at most \( |V_{X_x}| \). Let \( V' = V \setminus (\text{DAn}(x) \cup \text{Desc}(x)) \).

From \((X_x, Y_x)\), form sets \( X' \) and \( Y' \) as follows:

- \( X' = \{s\} \cup \{u_h, u_t\mid u \in (X_x \setminus V_{X_x}) \cap V'\} \cup \{u_h\mid u \in V_{X_x}\} \).
- \( Y' = \{t\} \cup \{u_h, u_t\mid u \in Y_x \cap V'\} \cup \{u_t\mid u \in V_{X_x}\} \).
We will show that \((X', Y')\) is a valid edge-cut for \(G\) of cost \(|V_{X'}|\). Intuitively, \((X', Y')\) has been built in such a way that there are only type-1 cut-edges \((u_h, u_t) \in E_X\); and there is an edge \((u_h, u_t) \in E_X\) iff \(u \in V_{X'}\). From the construction of \(X'\) and \(Y'\), we have \(X' \cup Y' = V_m\) and \(X' \cap Y' = \emptyset\). Further,

- There are no edges from \(s\) to \(Y'\):
  This is because \(s\) is constructed by contracting \(\text{DAnc}(x)\). Specifically, from the construction of \(X'\), we have, if \(v \in X_x\) then \(v_h \in X'\). Also, \(\text{Anc}(x) \subseteq X_x\) and we have no edges from \(\text{DAnc}(x)\) to outside \(\text{Anc}(x)\). Hence, there are no edges from the source vertex \(s \in V_m\) to \(Y'\).

- There are no edges from \(X'\) to \(t\):
  This is because \(t\) is constructed by contracting \(\text{Desc}(x)\) and for any cut-vertex \(v\) of \(X_x\), \(v_t\) is added to \(Y'\). Hence, since \(\text{Desc}(x) \subseteq Y_x\) and \((v \in V_{X_x} \implies v_t \in Y')\), we have no edges from \(X'\) to the target vertex \(t \in V_m\).

- There are no edges from \(\{u_h, u_t|u \in (X_x \setminus V_{X_x}) \cap V'\}\) to \(Y'\), or from \(X'\) to \(\{u_h, u_t|u \in V_x \cap V'\}\):
  From construction of \(G_m\), for any two pairs \(p_1 =< u_h, u_t >\) and \(p_2 =< v_h, v_t >\), that correspond to \(u, v \in V\), there are edges between vertices in \(p_1\) and \(p_2\) iff there is an edge \((u, v) \in E\). Hence, it follows from the fact that there are no edges from vertices in the set \((X_x \setminus V_{X_x})\) to \(Y_x\) and vice-versa.

- There are exactly \(|V_{X_x}|\) edges of weight one from \(\{u_h|u \in V_{X_x}\}\) to \(\{u_t|u \in V_{X_x}\}\):
  Corresponding to each cut-vertex \(v \in V_{X_x}\), we have a type-1 cut-edge \((v_h, v_t)\) in \((X', Y')\).

Hence, \((X', Y')\) forms a valid cut for \(G_m\) of cost \(|V_{X_x}|\).
Now, we will show that \(|V_{X_x}| \leq W_m(E_X)\) by constructing a convex-cut \(X_x = (X_x', Y_x')\) for \(G\) from the edge-min-cut \(X = (X, Y)\) with vertex-cut-size \(|V_{X_x}|\) at most \(W_m(E_X)\). We will do this in two parts: (1) As shown later, from \(X = (X, Y)\), we will first create an equivalent cut \(X' = (X', Y')\) for \(G_m\), of cost at most \(W_m(E_X)\), such that the its edge-cut-set \(E_{X'}\) contains only type-1 edges. (2) From \(X'\) it is easy to build a convex-cut \(X_x = (X_x', Y_x')\) as follows:

- \(X_x' = \text{DAnc}(x) \cup \{u|u_h \in X'\}\).
- \(Y_x' = \text{Desc}(x) \cup \{u|u_h \in Y'\}\).

Note that there are no edges from \(Y_x'\) to \(X_x'\), since if there was one such edge \((v, u)|v \in Y_x' \land u \in X_x'\), there would have been an edge \((u_h, v_h)\) of weight \(\infty\) from \(u_h \in X'\) to \(v_h \in Y'\), which is not possible as described earlier. From the construction of \(G_m\), \(X_x'\) and \(Y_x'\), it is easy to see that if there is an edge \((u, v)|u \in X_x' \land v \in Y_x'\), then there is necessarily a path \((u_h, u_t, v_h, v_t)|u_h \in X' \land v_h \in Y'\), and hence a type-1 edge from this path is in \(X'\). Since there are \(|E_{X'}|\) type-1 edges in \(X'\), we have at most \(|E_{X'}|\) cut-vertices in \(X_x\).

\(X' = (X', Y')\) is built from \(X = (X, Y)\) as follows: (Note that the edge-cut-set \(E_X\) will have no type-3 edges. So we only need to worry about removing type-2 edges from \(E_X\).)

- Start with \(X' = X\) and \(Y' = Y\).

- If \(E_X\) has an edge \((s, u_h)\) from the source, then move \(u_h\) from \(Y'\) to \(X'\), thus removing a type-2 edge from \(E_X\) and instead introducing at most one type-1 edge. Note that this step does not introduce any type-2 edge, since all the out-edges from a head node \(u_h\) are either of type-1 or type-3. Also, no type-3 edges
are introduced into $E_{X'}$ during this step, since we have a type-3 edge $(u_h, v_h)$ only if $(v, u) \in E$; since we have an edge from $s$ to $u_h$, we have $u \in \text{Anc}(x)$ and thus its predecessor $v \in \text{Anc}(x)$. Hence, $v_h$ was either already in $X'$ or $v_h$ will be eventually moved to $X'$ during this step.

- If $E_{X'}$ has any other type-2 edge which is of form $(u_t, v_h)$ or $(u_t, t)$, then $u_t$ can be moved from $X'$ to $Y'$, thus adding single type-1 edge $(u_h, u_t)$ to $E_{X'}$.

Hence, $(X'_x, Y'_x)$ forms a valid convex-cut for $G$ with vertex-cut-size at most $W_m(E_X)$.

\[\square\]

4.2 Heuristic for Min-Cut Based Lower Bounds

In this section, we will provide details on how a lower bound for an algorithm is obtained from a program in a completely automatic fashion. This process involves the following steps:

- Extracting CDAG from the input program.
- Decomposing the CDAG into multiple sub-CDAGs.
- Determining convex vertex-min-cut value (maximized over a set of vertices) of each sub-CDAG and finally adding them up (using Theorem 5) to obtain lower bound for the full CDAG.

Each of these steps are explained in detail below.
4.2.1 CDAG extraction

The very first step in our automated approach is to extract the implementation-independent CDAG from an input program. Automatically extracted CDAG contains a vertex for each instance of floating-point operation executed in the program during that particular run, and an edge for any read-after-write dependences between these floating-point operations. This process is done through LLVM [36] based dynamic analysis. It involves three major phases:

**Instrumentation:** Instrumenting the input program with function calls that allow us to capture runtime information.

**Execution:** Executing the instrumented program to obtain a trace file containing runtime details.

**Analysis:** Analyzing the trace file to generate CDAG.

All these steps are completely automatable using LLVM.

- The input program is initially compiled to LLVM intermediate representation (LLVM-IR). Our instrumentation function calls are inserted into this LLVM-IR through an optimization pass, and the instrumented LLVM-IR is compiled to a binary file.

- Instrumented binary file is executed to capture runtime details into a trace file. The generated trace file contains information about runtime program flow, i.e., information on which instructions were executed and in what order, which instruction instance wrote to which memory address, addresses from which the
operands were read, etc. This provides us sufficient details to build a CDAG for the algorithm corresponding to a program input.

- During analysis phase, events in the trace file are read in sequence. A vertex is created in CDAG for every floating-point event read from the trace file. In addition, a last-writer information is maintained to aid in creating edges. Last-writer information is a map from a memory address $A$ to the CDAG vertex $v$ that corresponds to the operation that recently wrote to $A$. Any subsequent floating-point operation (represented by vertex $w$ in CDAG) that reads from $A$ is data dependent on $v$. Hence, an edge is created from $v$ to $w$ in CDAG.

The actual implementation has several optimizations to minimize the generated trace file size and the memory footprint required for automatic extraction of CDAG. For instance, the trace file records events at basic block level instead of an individual instruction level since the instructions inside a basic block can be determined using static information once we know that the basic block was reached during runtime. These implementation details are not presented here. Complete details about instrumented function calls and trace file information can be found in [39, Chapter 2].

Note that the program execution order is not captured in the CDAG, thus making it independent of the schedule of the implementation from which we build the CDAG. We would also like to note that executed instructions and read/write memory addresses are dependent on problem size and program input, as is usually the case with any dynamic analysis based techniques. Hence, the extracted CDAG might vary in size and shape for different problem sizes and program inputs. We will show in Section 4.3 that, in spite of this, a fully automated approach for determining lower
bounds provides us insightful information about an algorithm, allowing us to derive useful conclusions. Also, for “regular” programs such as matrix-matrix multiplication, stencil computation, and several linear-algebra applications, graph structure and properties of the CDAG do not vary much between different inputs, allowing us to extrapolate and derive parametric lower bounds for such algorithms. Section 4.3.2 presents parametric lower bound results (in terms of decomposability factor) obtained through extrapolation for such algorithms.

4.2.2 Decomposition

Min-cut based lower bounding technique requires the CDAG to be decomposed and individually analyzed to obtain tighter bounds. Our implementation uses METIS [31] graph partitioner to partition the DAG of the extracted CDAG. CDAG extracted from the previous step is recursively bisected and minimum cut heuristics is executed at each level of recursion, as explained in next subsection and in Algorithm 4.2.

4.2.3 Finding maximum min-cut

As shown in Section 4.1, a standard max-flow algorithm can be used to find convex min-cut in our DAG. We use push-relabel based max-flow implementation provided by Lemon graph library [34], whose worst case time complexity per min-cut is $O(n^2 \sqrt{m})$, where $n$ is the number of vertices and $m$ is the number of edges in the graph. In order to obtain tighter bounds, convex min-cut is run with respect to all the vertices in a sub-CDAG and the maximum of those is taken. When the CDAG is too huge, convex min-cut is computed with respect to a subset of vertices, evenly sampled throughout the CDAG. Since convex-cut with respect to each vertex is independent of others, they can be executed in parallel to reduce the running time of the heuristic. Also,
since the convex vertex-min-cut is independent of the red pebble count $S$, it is possible to run the heuristic just once per input problem size and extract lower bound results for different sizes of $S$. Algorithms 4.1 and 4.2 contain the pseudo-codes for finding lower bound of a CDAG using min-cut based approach. The function TransformDAG (at Line 7 in Algorithm 4.1) transforms the DAG of a (sub-)CDAG into a digraph as detailed in Section 4.1. The function Maxflow calls the standard max-flow algorithm on a transformed graph.

![Algorithm 4.1](image)

### 4.3 Experimental Results

#### 4.3.1 Assessment of effectiveness of automated analysis

In this and next subsection, we show the effectiveness of our automated tool by running the tool on the algorithms for which tight analytical lower bounds are
**Algorithm 4.2:** Algorithm to compute min-cut based LB of a CDAG through recursive bisection.

```plaintext
Function MincutLB(C)
  Input : (sub-)CDAG C = (I, V, E, O)
  Input : Red pebble count S
  Output: Lower bound Q
  Q ← 0;
  if |V| ≥ 2S then
    (G₁ = (V₁, E₁), G₂ = (V₂, E₂)) ← Bisect((V, E));
    C₁ ← (I ∩ V₁, V₁, E₁ ∩ V₁);
    C₂ ← (I ∩ V₂, V₂, E₂ ∩ V₂);
    Q₁ ← MincutLB(C₁, S);
    Q₂ ← MincutLB(C₂, S);
    Q ← max(Q, Q₁ + Q₂);
    Q' ← SubCDAGLB(C, S);
    Q ← max(Q, Q');
  return Q;
```

already known, and comparing the results from the tool against the analytically derived bounds.

**Diamond DAG:** Tight analytical lower bound for diamond DAG was derived in Section 3.3.1. We obtained a lower bound of \((N - 2S)^2 / S \approx N^2 / S\)

Figure 4.1 shows the lower bound generated by our automated heuristic for different values of \(S\), for a diamond DAG of size 200 × 200. The data movement lower bound from the above analytical expression is also plotted. The bound generated by the automated CDAG analysis is less tight than the lower bound obtained using analytical reasoning, but shows the same trend as a function of \(S\).

**Diamond-Chain:** Figure 4.2 shows a CDAG analyzed by Bilardi et al. [9]. It is a diamond DAG with \(N^2\) nodes composed with a linear chain graph of \(N^2\) nodes, with
Figure 4.1: Lower bound for Diamond DAG: Analytical versus automated CDAG analysis.

a dependence edge from each diamond node to a distinct chain node. The dependence edges are in reverse order: the last diamond vertex connects to the first chain vertex, the first diamond vertex connects to the last chain vertex, etc. It could be seen that the vertex-cut-size of the convex min-cut with respect to the terminal vertex of the diamond is $N^2$, thus providing an analytical lower bound of $2 \times (N^2 - S)$ for diamond-chain.

Figure 4.2: Diamond-Chain CDAG [9].
Figure 4.3: Lower bound for Diamond-Chain: Analytical verses automated CDAG analysis.

Figure 4.3 shows the lower bound generated by automated CDAG analysis, that exactly matches the analytical bound, for different values of $S$ for a diamond-chain of size $N = 200$.

Jacobi computation: We obtained an analytical lower bound for two-dimensional 9-point Jacobi computation in Subsection 3.3.3. We obtained a lower bound of $0.75n^2T/\sqrt{S}$ for an $n \times n$ dimensional problem with $T$ time steps.

Figure 4.4 shows the lower bound generated by our automated heuristic for different values of $S$ for the problem size of $20 \times 20$ for 20 time-steps. The bound generated by the automated analysis follows the same pattern as that of the analytical bounds and in fact, provides a tighter bound than the analytical bounds. This is because we make conservative arguments to simplify the analysis when deriving bounds through analytical reasoning.
4.3.2 Parametric bounds through function fitting

The \textit{decomposability factor} $\lambda(S)$ of an algorithm is defined as the ratio between the sequential time of the algorithm, $|V|$, and the minimum data movement cost, $Q$ when $S$ red pebbles are used. For many algorithms whose decomposability factor [27] has been analytically characterized, it takes the form $\lambda(S) = k \times S^\alpha$. The automated tool might be helpful in determining the values of $k$ and $\alpha$ for an algorithm. The tool was run on CDAGs from different algorithms and a linear regression function fit was done for $\log(S)$ against $\log(\lambda(S))$. The slope and the intercept obtained from this function fit were then used to obtain the values of $\alpha$ and $k$, respectively. The Table 4.1 shows the $\lambda(S)$ values for different algorithms. The first column names the algorithm, the second column represents the known asymptotic analytical lower bound and the last column shows the values obtained through the automated tool. It may be seen that the form obtained through empirical function fitting is very close in most cases to the known analytical exponent on $S$ for the decomposability factor. Thus, the tool
holds promise for characterizing algorithms even if it is dependent on the problem input and value of $S$, if accurate parametric function fitting via extrapolation is possible.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Analytical bounds</th>
<th>Experimental bounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jacobi 3D</td>
<td>$O(S^{0.33})$</td>
<td>$14.6 \times S^{0.3}$</td>
</tr>
<tr>
<td>Jacobi 2D</td>
<td>$O(S^{0.3})$</td>
<td>$8.76 \times S^{0.48}$</td>
</tr>
<tr>
<td>ADI</td>
<td>$O(1)$</td>
<td>$1.55 \times S^{0.04}$</td>
</tr>
<tr>
<td>Diamond DAG</td>
<td>$O(S)$</td>
<td>$3.46 \times S^{0.87}$</td>
</tr>
<tr>
<td>Diamond chain</td>
<td>$O(1)$</td>
<td>$0.99 \times S^{0.004}$</td>
</tr>
</tbody>
</table>

Table 4.1: Decomposability factors for different algorithms.

4.3.3 Use case 1: Algorithm comparison and optimization

In this subsection, we present an illustrative case study to describe one approach to using lower bounds analysis to identify potential data movement bottlenecks in algorithms/applications. Given an implementation of an algorithm, the broad approach is as follows:

- For various values of $S$, generate a) lower bounds on data movement costs, using the approach described in the previous section, and b) number of data movements required for the actual sequential schedule for the computation, assuming an oracular (furthest-in-future) memory management strategy for the $S$ fast memory locations. If the lower bounds are close to the amount of data movement required for the given schedule, the conclusion is that the algorithm is fundamentally limited by its minimal inherent data movement requirements, and efforts to improve the implementation (e.g., via loop transformations like
tiling) will be futile. The only option in this scenario is to seek new algorithms with lower data movement complexity.

- If the analysis from the first step reveals a significant gap between the lower bound curve and the data movement cost for the actual schedule, it is suggestive that improvements to the algorithm implementation might be feasible. At this point, better alternative valid schedules for the CDAG are sought, through careful manual analysis of the algorithm.

We applied the above methodology to different comparison-based sorting algorithms. In this subsection we report on results for two sorting algorithms: quicksort and odd-even sort. Quicksort has an average computational complexity of $O(n \log_2 n)$ to sort $n$ elements, while odd-even sort is has an asymptotically higher complexity of $O(n^2)$, but a lower scaling coefficient for the quadratic complexity term. So efficient sorting routines often use a combination of an $O(n \log_2 n)$ algorithm like quicksort, along with a routine like odd-even sort, and a threshold value of $n$ (usually set to a value between 16 and 20) for switching between the two algorithms in a recursive execution.

We were interested in comparing the data movement complexities of the two sorting algorithms. Figure 4.5 compares the data movement complexity of quicksort and odd-even sort as a function of $S$, normalized to the number of comparisons performed. On the same graph, we also see the normalized number of data movements required in executing the sorting algorithms – by playing the red-blue pebble game using the code’s sequential execution order for firing of CDAG vertices (therefore providing an upper bound on the data movement complexity). It may be seen that while odd-even sort has a higher normalized data access cost than quicksort for the code’s sequential execution order, the lower bounds from automated CDAG analysis reveal
Figure 4.5: Quicksort versus odd-even sort: Normalized lower bound and cost for sequential execution order.

a significantly lower bound for odd-even sort than quicksort. Upon examining the CDAG for odd-even sort more carefully, it became apparent that it was amenable to skewing followed by loop tiling to significantly enhance data locality. In contrast, for the recursive quicksort algorithm, we could not identify any opportunity for loop transformations that would improve data locality. We implemented a register-tiled version of odd-even sort and evaluated performance for different register-tile sizes. Figure 4.6 displays relative time for odd-even sort over quicksort as a function of array size (using Intel ICC, for int data type), for the standard version as well as the modified register-tiled version. For the original version, the cross-over was around 16, i.e., odd-even sort was more efficient for sorting fewer than 16 elements, while quicksort was more efficient for sorting more than 16 elements. But with the tiled version of odd-even sort, the cross-over point was much higher - around 110 elements. Since our automated approach is based on dynamic analysis, the results obtained through our analysis depends on the input used for each run. Hence the user has to
be cautious before making generalized conclusions from the results of the automated analysis. The experiment in this section was run multiple times with different random inputs. Boundary cases like fully sorted and reverse sorted inputs were also tested and we found similar trends.

**Figure 4.6:** Execution time for original and tiled odd-even sort relative to quicksort.

### 4.3.4 Use case 2: Machine balance requirements

A processor’s machine balance is the ratio of the peak memory bandwidth to the peak floating-point performance. Lower bounds analysis can help identify whether some algorithms are inherently memory-bandwidth bound irrespective of all possible optimizations.

Jacobi iteration and Conjugate Gradient (CG) are two alternate iterative methods for solving a system of simultaneous equations that generally arise from discretization of steady-state problems. In Figure 4.7, we show the lower bounds of the memory bandwidth required per floating-point operation by Jacobi and CG methods to solve a regular 2D problem for two different problem sizes, as a function of $S$. The horizontal
lines in the figure represent the machine balance numbers (peak memory bandwidth over theoretical peak double-precision GFLOPS) of several widely used current processors. As can be seen from the figure, the lower bound characterization for CG stays above the machine balance line, irrespective of the value of $S$. In contrast, the profile for Jacobi crosses and drops below the machine balance lines for high values of $S$. This shows that irrespective of any transformations for locality enhancement, the CG method will be unavoidably memory-bandwidth bound for problem sizes that cannot fit in cache. The lower bound data also shows that increasing cache capacity will not provide any benefit. The only way to improve performance will be to increase the memory bandwidth. On the other hand, for the Jacobi algorithm, the profile shows that the algorithm is not constrained to be memory-bandwidth bound. With a sufficiently large cache, locality enhancing transformations could be successful in enhancing performance. In practice, the Jacobi method is generally not a particularly attractive iterative method due to its very slow convergence property. However, Jacobi iterations can be effectively used for the smoothing steps in multigrid solvers. While a few steps of CG could also be used instead of a Jacobi smoother, and the computational cost of a CG is not much higher than a Jacobi step, the data movement complexity is considerably higher. The most effective algorithms for future architectures could be very different from the preferred algorithms on current systems, and lower bounds analysis can be very helpful in anticipating trade-offs for future systems. The development of new methods suited for future systems can thus be usefully guided by modeling the data movement complexity of algorithms through lower bounds analysis.
Applications spend most of their running time (performing both computation and data movement) in nested loops. Loop tiling is a common technique used to improve data locality and in turn minimize data movement in a program. Manually determining tileability of a complicated loop nests (with function calls in their loop bodies) is a tedious process. Automatic loop transformation techniques such as polyhedral techniques [41] are usually limited to affine programs, rendering them useless on non-affine real world applications. In this section, we use dynamic analysis techniques and our automated lower bounding tool to aid in the process of determining tileability potential of arbitrary loop nests.

For illustration, consider a perfectly nested loop of depth two with corresponding CDAG $C = (I, V, E, O)$. Consider any untiled execution of this loop nest. In other words, any valid reordering of operations within a single iteration of the outer loop is permitted, while no reordering across outer loop iterations is allowed. Let $P_u$ be
the pebbling sequence of $C$ for this untiled execution. In $P_u$, vertices corresponding to execution of any outer loop iteration $i$ are fired before the vertices corresponding any outer loop iteration $j > i$. Let $i$ be any outer loop iteration and $T_{bi}$ and $T_{ei}$ be time-stamps at the beginning and end of the execution of iteration $i$, respectively. Let $S$ denote the size of fast memory. If $L_{bi} \subset V$ represents the (live) set of vertices fired before $T_{bi}$ with at least one successor fired between time-stamps $T_{bi}$ and $T_{ei}$, then $P_u$ necessarily incurs a data movement cost of at least $|L_{bi}| - S$ during the execution of iteration $i$. Similarly, if $L_{ei} \subset V$ represents the set of vertices fired during iteration $i$ (i.e., between time-stamps $T_{bi}$ and $T_{ei}$) and have successors fired after $T_{ei}$, then $P_u$ necessarily incurs a data movement cost of at least $|L_{ei}| - S$ after time $T_{ei}$. Further, within iteration $i$, $P_u$ incurs a data movement cost of at least $LB(C_i, S)$, where $C_i$ is the induced CDAG of $C$ containing vertices fired during iteration $i$ and $LB(C_i, S)$ represents a data movement lower bound for $C_i$ with $S$ red pebbles. Hence, the data movement cost $q(P_u, S)$ of any “untiled” pebbling sequence $P_u$ where tiling is disallowed satisfies

$$q(P_u, S) \geq T \sum_{i=1}^{T} (|L_{bi}| - S) + (|L_{ei}| - S) + LB(C_i, S).$$

where, $T$ represents the number of outer loop iterations.

Now, let $P_t$ represent any unrestricted pebbling sequence for $C$ (where tiling is permitted). $LB(C, S)$ is a data movement lower bound for $P_t$. If $LB(C, S) \geq \epsilon \times \sum_{i=1}^{T} (|L_{bi}| - S) + (|L_{ei}| - S) + LB(C_i, S)$, for some empirically chosen value of $\epsilon$ such as $\epsilon \geq 0.2$, then we can assert with high probability that tiling is not possible, or will not have much impact on the data movement cost of the loop nest. However, if $LB(C, S) \ll \sum_{i=1}^{T} (|L_{bi}| - S) + (|L_{ei}| - S) + LB(C_i, S)$, then there is a chance that tiling
would be beneficial for the loop nest, suggesting that it is worth spending manual effort looking for tiling possibilities.

This technique is also applicable to determine the potential fusing two loop nests. Let \( C \) be the CDAG of a program segment containing two consecutive loop nests. Let \( C1 = (I_1, V_1, E_1, O_1) \) and \( C2 = (I_2, V_2, E_2, O_2) \) be the sub-CDAGs of the two loop nests separately. Let \( \mathcal{P}_u \) be any pebbling sequence where fusion of the two loops are disallowed, i.e., vertices in \( V_1 \) are fired before vertices in \( V_2 \) in \( \mathcal{P}_u \). Let \( \mathcal{T}_1 \) denote the time-stamp at the end of execution of first loop nest in \( \mathcal{P}_u \). If \( L_1 \subseteq V_1 \) represents the set of vertices that were fired before \( \mathcal{T}_1 \) and have at least one successor fired after \( \mathcal{T}_1 \), then using the technique described above, if \( \text{LB}(C, S) \geq \epsilon \times (\text{LB}(C_1, S) + (|L_1| - S) + \text{LB}(C_2, S)) \), then it is safer to assume that fusing the two loops will not provide huge benefits, or else, a closer look at the code is necessary to determine if loop fusion is possible.

In general, for any imperfectly nested loops of any depth, this process is methodically performed as shown in Algorithm 4.3. Algorithm 4.4 returns True if two consecutive loop nests have potential for fusion. If Algorithm 4.4 returns True for a sequence of consecutive loops, there is a possibility to fuse them all together. For example, if \( L_1, L_2 \) and \( L_3 \) are three consecutive loop nests, and if Algorithm 4.4 finds potential in fusing \( L_1 \) and \( L_2 \), and \( L_2 \) and \( L_3 \), then it is possible to fuse \( L_1, L_2 \) and \( L_3 \) together. Algorithm 4.5 returns True if there is a potential in tiling a loop \( L \) and its immediate inner loops. If Algorithm 4.5 finds potential in a sequence of inner loops, then there is a possibility to tile them together. For example, if a program segment contains a loop nest of depth three containing loops \( L_1, L_2 \) and \( L_3 \) nested in this order, and if Algorithm 4.5 returns True for \( L_2 \), it indicates that loops \( L_2 \) and \( L_3 \) can
be possibly tiled using two-dimensional tiles. Further, if Algorithm 4.5 returns True for $L_1$, then all $L_1$, $L_2$ and $L_3$ can be possibly tiled using three-dimensional tiles. The function $\text{GetCDAG}(L, [a, b])$ returns the (sub-)CDAG corresponding to execution of iterations $a$ to $b$ (inclusive) of loop $L$.

<table>
<thead>
<tr>
<th>Function GetCDAG($L$, $[a, b]$) returns the (sub-)CDAG corresponding to execution of iterations $a$ to $b$ (inclusive) of loop $L$.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: Loop nest $L$; CDAG of $L$, $C = (I, V, E, O)$; Maximum loop depth $D$; Red pebble count $S$</td>
</tr>
<tr>
<td><strong>Data</strong>: $L.\text{numIters}$ is the number of runtime iterations of loop $L$</td>
</tr>
<tr>
<td><strong>Data</strong>: $L.\text{id}$ is a unique ID identifying loop $L$.</td>
</tr>
<tr>
<td><strong>Function</strong> $\text{CanFuseTile}(L, C, S)$</td>
</tr>
<tr>
<td>for $d = D - 1$ to 1 do</td>
</tr>
<tr>
<td>foreach loop $L$ in $L$ at depth $d$ do</td>
</tr>
<tr>
<td>$(L_1, \ldots, L_n) \leftarrow$ Sequence of loop nests nested immediately inside $L$;</td>
</tr>
<tr>
<td>for $i = 2$ to $n$ do</td>
</tr>
<tr>
<td>$C_1 = (I_1, V_1, E_1, O_1) \leftarrow \text{GetCDAG}(L_{i-1}, [1, L_{i-1}.\text{numIters}])$;</td>
</tr>
<tr>
<td>$C_2 = (I_2, V_2, E_2, O_2) \leftarrow \text{GetCDAG}(L_i, [1, L_i.\text{numIters}])$;</td>
</tr>
<tr>
<td>$C \leftarrow (I_1 \cup I_2, V_1 \cup V_2, V_1 \times V_2 \cap E, O_1 \cup O_2)$;</td>
</tr>
<tr>
<td>if $\text{CheckFuseable}(C, C_1, C_2, S) = \text{True}$ then</td>
</tr>
<tr>
<td>Print(“Loops ” + $L_{i-1}.\text{id}$ + “ and ” + $L_i.\text{id}$ “ are fuseable.”);</td>
</tr>
<tr>
<td>if $\text{CheckTileable}(L, S) = \text{True}$ then</td>
</tr>
<tr>
<td>Print(“Loop ” + $L.\text{id}$ + “ and its immediate inner loops are tileable.”);</td>
</tr>
</tbody>
</table>

| **Algorithm 4.3**: Checks fuseability and tileability potential of loop nests. |

Our automated approach requires full CDAG of a program to be held in the memory for analysis. CDAGs extracted from real world applications might contain several millions of vertices, making it infeasible to build the complete CDAG. Further, the trace file generated from the execution of an instrumented code can grow into several GBs when recording the events of the applications running for several minutes. However, many scientific computing applications are fairly regular in the sense that their
Input : CDAGs \( C = (I, V, E, O), C_1 = (I_1, V_1, E_1, O_1), C_2 = (I_2, V_2, E_2, O_2); \) 
Red pebble count \( S \)
Output: True if \( C_1 \) and \( C_2 \) are fuseable, False otherwise

Function \( \text{CheckFuseable}(C, C_1, C_2, S) \)

1. \( LB \leftarrow \text{GetLB}(C, S); \)
2. \( LB_1 \leftarrow \text{GetLB}(C_1, S); \)
3. \( LB_2 \leftarrow \text{GetLB}(C_2, S); \)
4. \( V' \leftarrow \{ v | v \in V_1 \land \exists w [(v, w) \in E \land w \in V_2] \}; \)
5. \( LB' \leftarrow LB_1 + LB_2 + |V'| - S; \)
6. if \( LB \geq \epsilon \times LB' \) then
7.  \( \text{return False} \)
8. else
9.  \( \text{return True} \)

Algorithm 4.4: Checks fuseability of two loop nests.

runtime data dependences do not show high variations between iterations. Hence, by analyzing a smaller sample of iteration domain, we will be able to draw useful conclusions about the complete execution. The following subsection details this process of sampling, where only a portion of execution details (based on memory availability) are recorded into the trace file and a sub-CDAG corresponding to this portion is extracted for analysis.

4.4.1 CDAG sampling

In this subsection, we are interested in extracting sub-CDAG corresponding to a portion of execution of a program, without having to record events for full execution. Particularly, we are interested in extracting sub-CDAG corresponding to first few iterations of each loop nest. For example, consider a simple perfectly nested two-dimensional loop nest, whose runtime trip-counts are 100 each, i.e., the iteration
domain of this loop nest is \{(i, j) | 1 \leq i, j \leq 100\}. We would like to extract sub-
CDAG corresponding to the sub-domain \{(i, j) | 1 \leq i, j \leq 10\}.

It is valid to under-approximate the sub-CDAG by ignoring some of the depend-
ence edges for our lower bounding purposes, but care should be taken to not include
any spurious edges. In Subsection 4.2.1, we noted that extraction of a CDAG in-
volves three phases – instrumentation, execution and analysis. To perform sampling,
following modifications are made to these three phases.

**Instrumentation:** Trip counters are added to loop headers of each loop to keep
track of iteration count during runtime. At the beginning of each iteration of a loop
during runtime, we need to detect whether we are inside or outside the sub-domain of interest. This is done by inserting a boolean variable and necessary conditional statements to loop headers. At runtime, this boolean variable is set (resp. unset) if the execution is inside (resp. outside) the sample sub-domain. In case of function calls, these states have to be properly propagated forward. For simplicity, these implementation details are not presented here.

**Execution & analysis:** As we can recall from Subsection 4.2.1, last-writer information is used during analysis phase to create CDAG edges. Hence, it is not sufficient to just record the events within the sample sub-domain, as this may result in spurious edges being created in a sub-CDAG. For instance, ignoring any write to a memory address outside the sample region might lead to an outdated last-writer map, introducing wrong data dependence edges. To prevent this, any address $A$ written outside sample region is recorded as invalid in trace file. During analysis phase, when an invalid-address event for $A$ is read from trace file, the entry for address $A$ in last-writer map is correspondingly marked as invalid and any operand that is read from an invalid address is ignored; no edge is created corresponding to this data dependence.

Within one continuous interval of execution outside sample region, it is sufficient to mark an address as invalid just once. Following a write to an address $A$ inside sample region, for the very first time an operation outside sample region overwrites $A$, address $A$ is recorded as invalid in trace file. $A$ is not recorded as invalid again until another operation inside sample region overwrites $A$.

We claim that this trace file information is sufficient to prevent any spurious edge from being created by our algorithm for extracting sub-CDAGs. As a contradiction,
assume that a spurious edge \( e = (u, v) \) is created in a CDAG. Our algorithm creates a vertex only for operations executed inside sample region, since no operation executed outside the sample region is stored in the trace file. Also, an edge \( (u', v') \) is created if and only if last-writer has an entry mapping a memory address \( A' \) to \( u' \), and \( v' \) corresponds to an operation inside sample region and reads from address \( A' \). Further an entry \( A' \rightarrow u' \) is made in last-writer only if \( u' \) stores its computed value into \( A' \). Let \( A \) be the memory address associated with data dependence captured by \( e \). Presence of edge \( (u, v) \) indicates that \( u \) and \( v \) are vertices for operations executed inside sample region, \( v \) reads from \( A \), and \( A \) is mapped to \( u \) and marked as valid during the time of creation of vertex \( v \). \( e \) being spurious indicates that there is another operation \( o \) between \( u \) and \( v \) that writes to \( A \). If \( o \) was inside sample region, then a vertex \( w \) would have been created in CDAG and entry \( A \rightarrow w \) would have been placed overwriting \( A \rightarrow u \) in last-writer map, thus preventing any edge from being created from \( u \) to \( v \). Hence \( o \) has to be outside the sample region. Since \( o \) writes to \( A \), there has to be at least one entry in trace file invalidating \( A \) after operation associated with \( u \) was executed. When this entry was read from trace file, entry \( A \rightarrow u \) would have been marked as invalid, contradicting our assumption that entry for \( A \) is valid in last-writer when vertex \( v \) is created.

4.4.2 Experimental results

We used our automated approach for analysis of SPEC benchmarks [25]. The benchmarks were profiled using HPCToolkit [28] to detect hot-loops. Table 4.2 shows the \( \epsilon \) values, i.e. ratio of the lower bounds for unrestricted and restricted schedules, of the outermost loops (i.e., time dimension) of the hot-loop nests for two of the
SPEC benchmarks. 410.bwaves is a computational fluid dynamics application whose implementation consists of an unfactored solver for implicit solution of Navier-Stokes equations using the Bi-CGstab algorithm. 459.GemsFDTD is a computational electromagnetics benchmark which solves 3D Maxwell equations using the finite-difference time-domain (FDTD) method. The two benchmarks show opposite trends. The ratio for 410.bwaves is much higher than that of 459.GemsFDTD. Further, the ratio remains almost constant for 410.bwaves with increasing pebble count, while we see a reduction in the ratio for 459.GemsFDTD with increasing pebble count. This signifies that 459.GemsFDTD might be time-tileable whereas there might not be a possibility to time-tile the main loop nest of 410.bwaves.

### Table 4.2: Ratios of lower bounds for unrestricted and restricted schedules for two SPEC benchmarks.

<table>
<thead>
<tr>
<th>Red pebbles, $S$</th>
<th>410.bwaves</th>
<th>459.GemsFDTD</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>0.437</td>
<td>0.037</td>
</tr>
<tr>
<td>1024</td>
<td>0.437</td>
<td>0.032</td>
</tr>
<tr>
<td>2048</td>
<td>0.437</td>
<td>0.027</td>
</tr>
<tr>
<td>4096</td>
<td>0.436</td>
<td>0.019</td>
</tr>
<tr>
<td>8192</td>
<td>0.435</td>
<td>0.012</td>
</tr>
</tbody>
</table>

4.5 Related Work

Several of the past works have mainly focused on deriving data movement lower bounds using problem specific analytical reasoning through manual application of the lower bounding techniques. Christ et al. [15] developed an approach to automatically derive lower bounds for a sub-class of computations containing indivisible loop body of
affine statements within a perfected nested loop. However, the techniques developed in this chapter can be applied on any arbitrary computations. We are unaware of any previous work that permits automatic handling of arbitrary computations for their lower bounds analysis.
CHAPTER 5

S-Partitioning for Lower Bounds Without Recomputation

In Chapter 3 we developed a graph min-cut based lower bounding technique for RB\textsubscript{NR} model and presented an automated heuristics to compute lower bounds. Here, we will present an adaptation of Hong & Kung’s S-partitioning technique for RB\textsubscript{NR} model. This modification allows us to formulate the problem of S-partitioning as an integer linear programming problem, thus enabling us to derive automated bounds for different algorithms using S-partitioning technique. Further, we will make use of some of the results developed in this chapter to derive parametric lower bounds for a sub-class of computations, called affine computations, in Chapter 6.

5.1 \textit{S}_{NR}-Partition

In this section, we introduce \textit{S}_{NR}-partition, which is an adaptation of Hong & Kung’s S-partition for RB\textsubscript{NR} model.

\textbf{Definition 25} (In-set). \textit{Given a CDAG }$C = (I,V,E,O)$\textit{ and a vertex-set }$V_s \subseteq V$, \textit{in-set of }$V_s$, $\text{ln}(V_s)$, \textit{is the set of vertices in }$V \setminus V_s$\textit{ that have at least one successor in }$V_s$. 

Definition 26 (Out-set). Given a CDAG $C = (I, V, E, O)$ and a vertex-set $V_s \subseteq V$, out-set of $V_s$, $\text{Out}(V_s)$, is the set of vertices in $V_s$ that have at least one successor outside $V_s$.

Definition 27 ($S_{NR}$-partition). Given a CDAG $C = (I, V, E, O)$, an $S_{NR}$-partition of $C$ is a sequence of $h$ subsets of $V$, $(V_1, \ldots, V_h)$, that satisfies the following properties:

P1 $i \neq j \implies V_i \cap V_j = \emptyset$ and $\bigcup_{i=1}^{h} V_i = V$.

P2 There is no cyclic dependence between the subsets.

P3 For every subset $V_i, 1 \leq i \leq h$, size of its in-set, $|\text{In}(V_i)| \leq S$.

P4 For every subset $V_i, 1 \leq i \leq h$, size of its out-set, $|\text{Out}(V_i)| \leq S$.

We provide a proof for Theorem 4 under RB$_{NR}$ game model below.

Theorem 14. Any pebbling sequence $\mathcal{P}$ obtained by playing the RB$_{NR}$ pebble game on a CDAG $C = (I, V, E, O)$ using at most $S$ red pebbles is associated with a $2S_{NR}$-partition $\mathcal{H}$ of $C$ such that $S \times h \geq q(\mathcal{P}, S) \geq S \times (h - 1)$, where $q(\mathcal{P}, S)$ is the cost of $\mathcal{P}$ and $h = |\mathcal{H}|$.

Proof. Consider a pebbling sequence $\mathcal{P}$ of RB$_{NR}$ pebble game containing $q = q(\mathcal{P}, S)$ pebblings with rules R1 or R2. Let $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_h)$ correspond to the sequence of pebbling subsequences obtained by partitioning $\mathcal{P}$ into $h = \lceil q/S \rceil$ consecutive subsequences, such that each $\mathcal{P}_i$, for $i = 1, \ldots, h - 1$, contains exactly $S$ pebblings with rules R1 or R2 and $\mathcal{P}_h$ contains no more than $S$ pebblings with rules R1 or R2.

Let $V_i \subseteq V, i = 1, \ldots, h$, be the set of vertices fired in the pebbling subsequence $\mathcal{P}_i$ (using rule R3$_{NR}$). We will show that $\mathcal{H} = (V_1, \ldots, V_h)$ is a $2S_{NR}$-partition of $C$ such that $S \times h \geq q(\mathcal{P}, S) \geq S \times (h - 1)$.
Since every vertex \( v \in V \setminus I \) is fired exactly once in \( \mathcal{P} \), property P1 of Definition 27 is trivially satisfied for \( \mathcal{H} \).

Since firing of any vertex \( v \in V \) in \( \mathcal{P} \) is possible only after all its predecessors have been fired (so as to satisfy rule R3\(_{NR}\)), any predecessor, \( p \), of \( v \) should have been necessarily fired in some pebbling subsequence \( \mathcal{P}_j \), \( j \leq i \), and hence, \( p \in V_j \), \( j \leq i \), satisfying property P2.

To prove satisfiability of property P3, for a given \( V_i \), consider the following two sets: (1) \( V_i^{r0} \): set of vertices that have a red pebble on them just before the beginning of the pebbling subsequence \( \mathcal{P}_i \) in \( \mathcal{P} \); (2) \( V_i^{br} \): set of vertices on which a red pebble is placed according to rule R1 in \( \mathcal{P}_i \). We have, the in-set of \( V_i \), \( \text{In}(V_i) \subseteq V_i^{r0} \cup V_i^{br} \). Thus \( |\text{In}(V_i)| \leq |V_i^{r0}| + |V_i^{br}| \). As there only \( S \) red pebbles, \( |V_i^{r0}| \leq S \). Also by construction of \( \mathcal{P}_i \), \( |V_i^{br}| \leq S \). Hence, \( |\text{In}(V_i)| \leq 2S \).

Satisfiability of property P4 by \( \mathcal{H} \) is proved in the similar way: Let \( V_i^{r1} \) be the set of vertices that have a red pebble on them at the end of the pebbling subsequence \( \mathcal{P}_i \) in \( \mathcal{P} \), and \( V_i^{rb} \) be the set of vertices of \( V_i \) on which a blue pebble is placed in \( \mathcal{P}_i \) according to rule R2. We have, the out-set of \( V_i \), \( \text{Out}(V_i) \subseteq V_i^{r1} \cup V_i^{rb} \). Thus \( |\text{Out}(V_i)| \leq |V_i^{r1}| + |V_i^{rb}| \). As there are only \( S \) red pebbles, \( |V_i^{r1}| \leq S \). Also by construction of \( \mathcal{P}_i \), \( |V_i^{rb}| \leq S \). Hence, \( |\text{Out}(V_i)| \leq 2S \). \( \Box \)

**Lemma 4.** Let \( H(2S) \) be the minimal number of vertex sets of any \( 2S_{NR} \)-partition of a CDAG \( C \). Then the data movement complexity of \( C \) without recomputation, \( Q \geq S \times (H(2S) - 1) \)
5.2 ILP Formulation for $S_{NR}$-Partitioning Based Lower Bounds

Size of the smallest $2S_{NR}$-partition of a CDAG $C = (I, V, E, O)$ can be lower bounded by the size of the largest vertex set in any $2S_{NR}$-partition of $C$ as follows: Let $H$ denote the number of vertex sets in a smallest $2S_{NR}$-partition, $\mathcal{H} = (V_1, V_2, \ldots, V_H)$, of $C$. Let $V_x$ be the largest vertex set in $\mathcal{H}$, i.e., $x = \arg\max_i |V_i|$, where, $i = 1, \ldots, H$. Then, we have,

$$\left\lceil \frac{|V|}{|V_x|} \right\rceil \leq \left\lceil \frac{|V|}{|U|} \right\rceil \leq H$$

Hence, if $U$ is an over-approximation of $V_x$, then from Lemma 4, we have,

$$Q \geq S \times \left( \left\lceil \frac{|V|}{|U|} \right\rceil - 1 \right) \quad (5.1)$$

For a given CDAG $C = (I, V, E, O)$, the following integer linear programming (ILP) formulation provides an upper bound for the size of the largest vertex set $|U|$.

Maximize $\sum_{v \in V} x_v$
subject to $\forall (u, v) \in E$, $x_u + y_u \geq x_v$
$\forall v \in I$, $x_v = 0$ \hspace{1cm} (5.2)

$\sum_{v \in V} y_v \leq 2S$
$\forall v \in V$, $0 \leq x_v, y_v \leq 1$

Each vertex $v \in V$ is associated with two boolean variables $x_v$ and $y_v$. $x_v$ determines if $v$ belongs to the over-approximated vertex set $U$ or not. $y_v$ determines whether $v$ belongs to the in-set of $U$. The maximization is performed under the constraints that if a vertex $v$ belongs to $U$, then its predecessor $u$ should either be in $U$ or be in the in-set of $U$. The second constraint prevents inputs from being part of $U$ and the third constraint restricts the size of the in-set to be less than or equal to $2S$.  

82
The above ILP formulation can be used to automatically determine a lower bound of any arbitrary CDAG. The process of automatically extracting the CDAG of an algorithm was described in Subsection 4.2.1.

5.3 Experimental Results

The combinatorial nature of integer programming problem restricts the size of the CDAGs that could be analyzed using $S$-partitioning based automated approach. However, in certain cases, especially when CDAGs have “regular” structure, it is possible to draw useful conclusions based on the automated analysis on smaller problem sizes. As in Section 4.3, we will first show the effectiveness of our automated approach by comparing the results against previously know analytical lower bounds.

Irony et al. [29] showed that the data movement complexity $Q$ for $N \times N$ matrix-matrix multiplication (matmult) satisfies $Q \geq \frac{N^3}{(2\sqrt{2})} - S$. Figure 5.1 shows the automated analysis results for matmult for a problem size of $N = 30$. The bound

![Graph showing lower bound for matmult: Analytical versus automated CDAG analysis.]

**Figure 5.1:** Lower bound for matmult: Analytical versus automated CDAG analysis.
generated by the automated analysis follows the same trend as that of the analytica bounds. As seen for the case of Jacobi 2D in Subsection 4.3.1, the automated analysis for matmult provides us a tighter bound.

5.3.1 Use case: Compiler evaluation

A potential use of obtaining lower bounds on data movement complexity of computations is in evaluating the effectiveness of automatic transformations generated by compilers. It can help compiler writers identify segments of code that may not have been optimally transformed by the compiler. We demonstrate this below using matrix-matrix multiplication as an example. Pluto [41] is a fully automatic polyhedral based optimizing compiler. Figure 5.2 shows the data movement lower bound from our automated analysis against the data movement cost of the schedule produced by Pluto-tiled code for a $10 \times 10$ matrix for various red pebble counts that correspond to different register sizes. (Different tile sizes were experimented to find the best tile size for Pluto-tiled code).

![Figure 5.2: Comparison of data movement cost of Pluto’s register-tiled schedule against the lower bound.](image)

84
Based on the sizeable gap between the two curves in Figure 5.2, the generated
tiled code from Pluto was studied. Listing 5.1 shows the register-tiled code generated
by Pluto. From a closer look at the code, it was clear that a changed loop order

```c
register int ra,rb,rc,rd,re,rf,rg,rh,ri,rj,rk,rl,rm;
for(it = 0; it < N; it+=3)
   for(jt = 0; jt < N; jt+=3)
      for(kt = 0; kt < N; kt+=3) {
         ra = B[kt][jt]; rb = B[kt][jt+1]; rc = B[kt][jt+2];
         rd = B[kt+1][jt]; re = B[kt+1][jt+1]; rf = B[kt+1][jt+2];
         rg = B[kt+2][jt]; rh = B[kt+2][jt+1]; ri = B[kt+2][jt+2];
         for(i = it; i < min(it+3,N); i++) {
            rj = A[i][kt]; rk = A[i][kt+1]; rl = A[i][kt+2];
            rm = C[i][jt];
            rm += (rj*ra); rm += (rk*rd); rm += (rl*rg);
            C[i][jt] = rm;
            rm = C[i][jt+1];
            rm += (rj*rb); rm += (rk*re); rm += (rl*rh);
            C[i][jt+1] = rm;
            rm = C[i][jt+2];
            rm += (rj*rc); rm += (rk*rf); rm += (rl*ri);
            C[i][jt+2] = rm;
         }
      }
```

**Listing 5.1:** Register-tiled code generated by Pluto

should improve performance. Pluto’s register-tiled code with manual permutation
applied is shown in Listing 5.2. Figure 5.3 shows the improvement in performance
from the modified code. This analysis provides clues to the compiler developer on the
optimizations that were missed by the compiler.
register int ra,rb,rc,rd,re,rf,rg,rh,ri,rj,rk,rl,rm;

for(jt = 0; jt < N; jt+=3) {
    for(kt = 0; kt < N; kt+=3) {
        ra = B[kt][jt]; rb = B[kt][jt+1]; rc = B[kt][jt+2];
        rd = B[kt+1][jt]; re = B[kt+1][jt+1]; rf = B[kt+1][jt+2];
        rg = B[kt+2][jt]; rh = B[kt+2][jt+1]; ri = B[kt+2][jt+2];
        for(it = 0; it < N; it+=3) {
            for(i = it; i < min(it+3,N); i++) {
                rj = A[i][kt]; rk = A[i][kt+1]; rl = A[i][kt+2];

                rm = C[i][jt];
                rm += (rj*ra); rm += (rk*rd); rm += (rl*rg);
                C[i][jt] = rm;

                rm = C[i][jt+1];
                rm += (rj*rb); rm += (rk*re); rm += (rl*rh);
                C[i][jt+1] = rm;

                rm = C[i][jt+2];
                rm += (rj*rc); rm += (rk*rf); rm += (rl*ri);
                C[i][jt+2] = rm;
            }
        }
    }
}

Listing 5.2: Register-tiled code generated by Pluto

Figure 5.3: Performance of register-tiled codes with Pluto’s original schedule and Pluto’s schedule with manual permutation applied.
CHAPTER 6

Parametric Lower Bounds for Affine Computations Using Static Analysis

In this chapter, we develop a static analysis approach to derive asymptotic parametric data movement lower bounds as a function of cache size and problem size for affine computations. Affine computations can be modeled using (union of) convex sets of integer points, and (union of) relations between these sets. The motivation is twofold. First, there exists an important class of affine computations whose control and data flow can be modeled exactly at compile-time using only affine forms of the loop iterators surrounding the computation statements, and program parameters (constants whose values are unknown at compile-time). Many dense linear algebra computations, image processing algorithms, finite difference methods, etc., belong to this class of programs [23]. Second, there exist readily available tools to perform complex geometric operations on such sets and relations. We use the Integer Set Library (ISL) [54] for our analysis.
6.1 Background

6.1.1 Polyhedron representation for affine codes

In the following, we use ISL terminology [54] and syntax to describe sets and relations. We now recall some key concepts to represent program features.

**Definition 28 (Polyhedron).** A polyhedron is an intersection of a finite number of half-spaces.

**Definition 29 (Z-polyhedron).** A Z-polyhedron is the intersection of a polyhedron with the integer lattice. In other words, a Z-polyhedron is the set of integer points inside a polyhedron.

**Iteration domain:** A computation vertex in a CDAG represents a dynamic instance of some operation in the input program. For example, given a statement

\[ S_1 : A[i] += B[i+1] \]

surrounded by one loop

\[ \text{for}(i = 0; i < n; ++i), \]

the operation \( += \) will be executed \( n \) times, and each such dynamic instance of the statement corresponds to a vertex in the CDAG. For affine programs, this set of dynamic instances can be compactly represented as a (union of) Z-polyhedra, i.e., a set of integer points bounded by affine inequalities intersected with an affine integer lattice [24]. Using ISL notation, the iteration domain of statement \( S_1 \), \( D_{S_1} \), is denoted:

\[ [n] \rightarrow \{ S_1[i] : 0 \leq i < n \}. \]

The left-hand side of \( \rightarrow \), \( [n] \) in the example, is the list of all parameters needed to define the set. \( S_1[i] \) models a set with one dimension ([i]) named \( i \), and the set space is named \( S_1 \). Presburger formulae are used on the right-hand side of “:” to model the points belonging to the set. In ISL, these sets are disjunctions of conjunctions of Presburger formulae, thereby modeling unions of
convex and strided integer sets. The **dimension** of a set $S$ is denoted as $\dim(S)$. $\dim(S1) = 1$ in the example above. The **cardinality** of set $S$ is denoted as $|S|$. $|S1| = n$ for the example. Standard operations on sets, such as union, intersection, projection along certain dimensions, are available. In addition, key operations for analysis, such as building counting polynomials for the set (i.e., polynomials of the program parameters that model how many integer points are contained in a set; $n$ in our example) [5], and parametric (integer) linear programming [21] are possible on such sets. These operations are available in ISL.

We remark that although our analysis relies on integer sets and their associated operations, it is not limited to programs that can be exactly captured using such sets (e.g., purely affine programs). Since we are interested in computing lower bounds on data movement complexity, an under-approximation of the statement domain and/or the set of dependences is acceptable, since a lower bound for the approximated system is a valid lower bound for the actual system. For instance, if the iteration domain $\mathcal{D}_S$ of a statement $S$ is not described exactly using Presburger formulae, we can under-approximate this set by taking the largest convex polyhedron $\overline{\mathcal{D}}_S \subseteq \mathcal{D}_S$. Such a polyhedron can be obtained, for instance, by first computing the convex hull $\overline{\mathcal{D}}_S \supseteq \mathcal{D}_S$ and then shifting its faces until they are strictly included in $\mathcal{D}_S$. We also remark that such sets can be extracted from an arbitrary CDAG (again using approximations) by means of trace analysis, and especially trace compression techniques for vertices modeling the same computation [32].

**Relations:** In the graph $G = (V, E)$ of a CDAG $C = (I, V, E, O)$, vertices are connected by producer-consumer edges capturing the data flow between operations.
Similar to iteration domains, affine forms are used to model the relations between the points in two sets. Such relations capture which data is accessed by a dynamic instance of a statement, as in classical data-flow analysis. In the example above, elements of array $B$ are read in statement $S_1$, and the relation $R_1$ describing this access is: $\{n\} \rightarrow \{S_1[i] \rightarrow B[i+1] : 0 \leq i < n\}$. This relation models a single edge between each element of set $S_1$ and an element of set $B$, described by the relationship $i \rightarrow i + 1$. Several operations on relations, such as $\text{domain}(R)$, which computes the domain (e.g., input set) of the relation ($\text{domain}(R_1) = \{n\} \rightarrow \{[i] : 0 \leq i < n\}$), $\text{image}(R)$ computing the image (e.g., range, or output set) of $R$ ($\text{image}(R_1) = \{n\} \rightarrow \{[i] : 1 \leq i < n+1\}$), the composition of two relations $R_1 \circ R_2$, their union $\cup$, intersection $\cap$, difference $\setminus$ and the transitive closure $R^+$ of a relation, are available.

All these operations are supported by ISL.

Relations can also be used to directly capture the connections between computation vertices. For instance, given two statements $S_1$ and $S_2$ with a producer-consumer relationship, the edges connecting each dynamic instance of $S_1$ and $S_2$ in a CDAG can be expressed using relations. For example, $\{n\} \rightarrow \{S_1[i,j] \rightarrow S_2[i,j-1,k] : \ldots\}$ models a relation between a 2D statement and a 3D statement. Each point in $S_1$ is connected to several points in $S_2$ along the $k$-dimension.

We note that in a similar manner to iteration domains for vertices, relations can also be extracted from non-affine programs via convex under-approximation or from the CDAG via trace analysis. Again, care must be taken to always properly under-approximate the relations capturing data dependences: it is safe to ignore a dependence (it can only lead to under-approximation of the data flow and therefore
the data movement requirement), and therefore we only consider must-dependences in our analysis framework.

**CDAG and Z-polyhedron:** The CDAG of an affine loop nest (i.e., a sequence of (im)perfectly nested loops whose loop bounds and array accesses are affine functions of surrounding loop variables and program parameters) is closely related to the (union of) Z-polyhedra representing the (union of) domains of its statements. This relationship is shown in Figure 6.1. Let \( Z \) denote the union of Z-polyhedra representing the union of statement domains in \( \mathbb{Z}^d \) space, where \( d \) is the maximum loop depth of the affine loop nest. Each vertex in the CDAG \( C \) is associated with an integer point in \( Z \) as follows: a vertex \( v \) in \( C \) that is computed during iteration \((i, j)\) is associated with the point \((i, j)^T \in Z\). The edges in \( C \) are aligned along the dependence directions in \( Z \).

**Flow-dependence graph:** A flow-dependence graph of a computation is a directed multi-graph. There is a vertex corresponding to each input and each statement of the program. Each vertex \( v \) has an associated domain \( D_v \), representing the domain of the corresponding statement. Edges between the vertices capture the flow-dependences between the statements. Each edge \( e \) has an associated relation, \( R_e \), representing the affine dependence relation between the source and target vertices of the edge.

### 6.1.2 Geometric inequalities

**Loomis-Whitney inequality [37]:** Loomis-Whitney inequality is a geometric result that allows to estimate the size of a \( d \)-dimensional set in terms of the sizes of its \((d-1)\)-dimensional orthogonal projections onto the coordinate hyperplanes. If \( E \subseteq \mathbb{R}^d \)
// Parameters: N, T
// Inputs: I[N]
// Outputs: A[N]
for (i=0; i<N; i++)
S1: A[i] = I[i];

for (t=1; t<T; t++)
{
    for (i=1; i<N-1; i++)
    for (i=1; i<N-1; i++)
S3: A[i] = B[i];
}

(a) Jacobi 1D stencil computation.

(b) CDAG.

(c) Flow-dependence.

Figure 6.1: Jacobi 1D computation, its CDAG and flow-dependence graph.

is some measurable set, and $\phi_j : \mathbb{R}^d \rightarrow \mathbb{R}^{d-1}$, for $1 \leq j \leq d$, are $d$ orthogonal projections onto coordinate hyperplanes, i.e., $\phi_j(x_1, \ldots, x_d) = (x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_d)$, then, by the Loomis-Whitney inequality, size (using Lebesgue measure) of the set $E$, $|E| \leq \prod_{j=1}^d |\phi_j(E)|^{1/(d-1)}$. For example, if $E$ is some set in $\mathbb{R}^3$ (refer Figure 6.2) and

92
$E_i$, $E_j$ and $E_k$ are the projections of $E$ along directions $i$, $j$ and $k$, respectively, then by the Loomis-Whitney inequality, volume of $E$, $|E| \leq \sqrt{|E_i| \times |E_j| \times |E_k|}$.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{loomis-whitney.png}
\caption{Illustration of Loomis-Whitney inequality in three-dimensional space. Here, volume of the red sphere is bounded from above by the product of square-root of the area of the three projected circles.}
\end{figure}

**Brascamp-Lieb inequality:** Brascamp-Lieb inequality is the generalization of the Loomis-Whitney inequality, where the orthogonal projections onto the coordinate hyperplanes are replaced by more general linear maps, not necessarily all mapping to the same dimension. (Refer Figure 6.3.) Christ et al. [15, Theorem 3.2] extended a discrete case of a Brascamp-Lieb inequality [6, Theorem 2.4] for deriving communication lower bounds under geometric model. The extension of the continuous Brascamp-Lieb inequality, in the restricted case of orthogonal projections and using
Lebesgue measure for the sizes of the sets, is stated below. We use the notation $H \leq \mathbb{R}^d$ to denote that $H$ is a linear subspace of $\mathbb{R}^d$.

**Theorem 15.** Let $\phi_j : \mathbb{R}^d \to \mathbb{R}^d$, be an orthogonal projection for $j \in \{1, 2, \ldots, m\}$ such that $\phi_j(x_1, \ldots, x_d) = (y_1, \ldots, y_d_j)$ where $\{y_1, \ldots, y_d_j\} \subseteq \{x_1, \ldots, x_d\}$.

Then, for $(s_1, \ldots, s_m) \in [0, 1]^m$:

$$\forall H \leq \mathbb{R}^d, \dim(H) \leq \sum_{j=1}^{m} s_j \dim(\phi_j(H)) \quad (6.1)$$

$$\iff \forall E \subseteq \mathbb{R}^d, |E| \leq \prod_{j=1}^{m} |\phi_j(E)|^{s_j} \quad (6.2)$$

![Figure 6.3: Illustration of Brascamp-Lieb inequality.](image)

In our restricted case, since the linear maps $\phi_j$ are orthogonal projections, the following Theorem enables us to limit the number of inequalities of (6.1) required for Theorem 15 to hold. Only one inequality per subspace $H_i$, defined as the linear span
of the canonical vector $e_i$, is required ($\langle e_i \rangle$ represents the subspace spanned by the vector with a non-zero only in the $i^{th}$ coordinate). The following theorem is equivalent to [15, Theorem 6.6]. (See also [6, Proposition 7.1].)

**Theorem 16.** Let $\phi_j : \mathbb{R}^d \to \mathbb{R}^d$ be an orthogonal projection for $j \in \{1, 2, \ldots, m\}$ such that $\phi_j(x_1, \ldots, x_d) = (y_1, \ldots, y_d)$ where $\{y_1, \ldots, y_d\} \subseteq \{x_1, \ldots, x_d\}$. Then, for $(s_1, \ldots, s_m) \in [0, 1]^m$:

$$\forall H \leq \mathbb{R}^d, \; \dim(H) \leq \sum_{j=1}^{m} s_j \dim(\phi_j(H)) \quad (6.3)$$

$$\iff \forall H_i = \langle e_i \rangle, \; 1 = \dim(H_i) \leq \sum_{j=1}^{m} s_j \delta_{i,j} \quad (6.4)$$

where, $\delta_{i,j} = \dim(\phi_j(H_i))$

6.2 Brascamp-Lieb Inequality and $2S_{NR}$-Partitioning Based Lower Bound

Figure 6.4 shows a simple loop nest with a single statement and its corresponding CDAG as a $\mathbb{Z}$-polyhedron. The integer points on the coordinate line $\{i = 0\}$ represent the input vertices of array $B$ and the integer points on the coordinate line $\{j = 0\}$ represent the input vertices of array $A$. An integer point $(i, j) \in \mathbb{Z}^2$ corresponds to the vertex in the CDAG computed at iteration $(i, j)$. For convenience, in the descriptions below, a vertex in a CDAG is sometimes directly referred to by its corresponding integer point in the $\mathbb{Z}$-polyhedron. Since an operation during iteration $(i, j)$ reads values from elements $A[i]$ and $B[j]$, images of the projection of a point $p \in \mathcal{D}_{S1}$ along the dependence directions $(1, 0)^T$ and $(0, 1)^T$ provide vertices in its in-set, $\mathbf{in}(\{p\})$. 95
Hence, if $E \subseteq D_{S_1}, \phi_1 : (i, j) \rightarrow i$ and $\phi_2 : (i, j) \rightarrow j$, then $\phi_1(E) \subseteq \ln(E)$ and $\phi_2(E) \subseteq \ln(E)$.

for $i=0; i<N; i++$
for $j=0; j<N; j++$

Figure 6.4: Example to illustrate the relationship between geometric inequality and data movement lower bounds.

From the definition of a $S_{NR}$-partition, any vertex-set $V_i$ in a $2S_{NR}$-partition has the property $|\ln(V_i)| \leq 2S$. Hence, in our example, we have $|\phi_1(V_i)| \leq 2S$ and $|\phi_2(V_i)| \leq 2S$. The Brascamp-Lieb inequality allows us to upper bound the size of any set $E \subseteq \mathbb{R}^d$ in terms of the sizes of the images of its projections. Hence, size of the largest vertex-set $U$ of any $2S_{NR}$-partition, in our example, is bounded as follows:

$|U| \leq \prod_{j=1}^{2} |\phi_j(U)|^{s_j} \leq \prod_{j=1}^{2} (2S)^{s_j} = (2S)^{\sum_{j=1}^{2} s_j}$, for $s = (s_1, s_2) \in [0, 1]^2$, subject to the constraint $\forall H_i, 1 \leq \sum_{j=1}^{2} s_j \delta_{i,j}$. In general, we have, size of the largest vertex-set $|U| \leq \prod_{j=1}^{m} (2S)^{s_j} = (2S)^{\sum_{j=1}^{m} s_j}$, for $s = (s_1, \ldots, s_m) \in [0, 1]^m$, subject to the constraint $\forall H_i, 1 \leq \sum_{j=1}^{m} s_j \delta_{i,j}$. In order to obtain as tight asymptotic bound as possible, we seek to find $s$ such that $|U|$ is as small as possible. This corresponds to finding $s_j$ such that $\prod_{j=1}^{m} (2S)^{s_j}$ is minimized, or equivalently, $(2S)^{\sum_{j=1}^{m} s_j}$ is minimized.

96
For this purpose, if \( \forall i, \exists j \), s.t., \( \delta_{i,j} = 1 \), we solve:

\[
\text{Minimize } \sum_{j=1}^{m} s_j, \quad \text{s.t., } \forall i, \ 1 \leq \sum_{j=1}^{m} s_j \delta_{i,j} \quad (6.5)
\]

We can instead solve the following dual problem, whose solution gives an indication of the shape of the optimal “cube.”

\[
\text{Maximize } \sum_{i=1}^{d} x_i, \quad \text{s.t., } \forall j, \ \sum_{i=1}^{d} x_i \delta_{i,j} \leq 1 \quad (6.6)
\]

In our example, we have:

\[
\phi_1 : (i, j) \rightarrow (i); \ \phi_2 : (i, j) \rightarrow (j)
\]

Let \( H_1 \) and \( H_2 \) denote the two subspaces spanned by the vectors \( (1, 0)^T \) and \( (0, 1)^T \), respectively. We have \( \delta_{1,1} = \dim(\phi_1(H_1)) = 1 \) and \( \delta_{2,1} = \dim(\phi_1(H_2)) = 0 \). This provides us the first constraint for (6.6) as \( x_1.1 + x_2.0 \leq 1 \) or \( x_1 \leq 1 \). Similarly, \( \delta_{1,2} = 0 \) and \( \delta_{2,2} = 1 \) providing us the second constraint \( x_2 \leq 1 \). This results in the following linear programming problem:

\[
\text{Maximize } x_1 + x_2 \quad (6.7)
\]

s.t. \( x_1 \leq 1; \ x_2 \leq 1 \)

Solving (6.7) provides the solution \( x_1 = x_2 = 1 \), or, \( x_1 + x_2 = 2 \). Hence, size of the largest vertex-set \( |U| = O(S^{x_1+x_2}) = O(S^2) \). Hence, we obtain a data movement lower bound of \( Q = \Omega \left( N^2/S \right) \) (using the Inequality (5.1)) for our example.

In the next section, we detail methods to obtain valid projection directions automatically through static analysis of an input affine code.
6.3 Automated Lower Bound Computation Through Static Analysis

We present a static analysis algorithm for automated derivation of expressions for parametric asymptotic data movement lower bounds for algorithms. We use two illustrative examples to explain various steps in the algorithm before providing detailed pseudo-code for the algorithm.

6.3.1 Illustrative example 1

Consider the example of Jacobi 1D stencil computation in Figure 6.1.

Statements and domains: Figure 6.1(c) shows the static flow-dependence graph $G_F = (V_F, E_F)$ for Jacobi 1D. $G_F$ contains a vertex for each statement in the program. The input array $I$ is also explicitly represented in $G_F$ by node $I$ (shaded in black in Figure 6.1(c)). Each vertex has an associated domain as shown below:

- $D_I = [N] \rightarrow \{I[i]: 0 \leq i < N\}$
- $D_{S1} = [N] \rightarrow \{S1[i]: 0 \leq i < N\}$
- $D_{S2} = [T, N] \rightarrow \{S2[t, i]: 1 \leq t < T \text{ and } 1 \leq i < N-1\}$
- $D_{S3} = [T, N] \rightarrow \{S3[t, i]: 1 \leq t < T \text{ and } 1 \leq i < N-1\}$

Edges and relations: The edges represent true (read-after-write) data dependences between the statements. Each edge has an associated affine dependence relation as shown below:
- Edge $e_1$: This edge corresponds to the dependence due to copying the inputs $I$ to array $A$ at statement $S_1$ and has the following relation.
  \[
  R_{e_1} = \{N\} - \{I[i] \rightarrow S_1[i]: 0 \leq i < N\}
  \]

- Edges $e_2$, $e_3$ and $e_4$: The use of array elements $A[i-1]$, $A[i]$ and $A[i+1]$ at statement $S_2$ are captured by edges $e_2$, $e_3$ and $e_4$, respectively.
  \[
  R_{e_2} = \{T,N\} - \{S_1[i] \rightarrow S_2[1,i+1]: 1 \leq i < N-2\}
  \]
  \[
  R_{e_3} = \{T,N\} - \{S_1[i] \rightarrow S_2[1,i]: 1 \leq i < N-1\}
  \]
  \[
  R_{e_4} = \{T,N\} - \{S_1[i] \rightarrow S_2[1,i-1]: 2 \leq i < N-1\}
  \]

- Edges $e_5$ and $e_6$: Multiple uses of the boundary elements $I[0]$ and $I[N-1]$ by $A[t][1]$ and $A[t][N-2]$, respectively, for $1 \leq t < T$ are represented by the following relations.
  \[
  R_{e_5} = \{T,N\} - \{S_1[0] \rightarrow S_2[t,1]: 1 \leq t \leq T\}
  \]
  \[
  R_{e_6} = \{T,N\} - \{S_1[N-1] \rightarrow S_2[t,N-2]: 1 \leq t \leq T\}
  \]

- Edge $e_7$: The use of array $B$ in statement $S_3$ corresponds to edge $e_7$ with the following relation.
  \[
  R_{e_7} = \{T,N\} - \{S_2[t,i] \rightarrow S_3[t,i]: 1 \leq t \leq T \text{ and } 1 \leq i < N-1\}
  \]

- Edges $e_8$, $e_9$ and $e_{10}$: The uses of array $A$ in statement $S_2$ from $S_3$ are represented by these edges with the following relations.
  \[
  R_{e_8} = \{T,N\} - \{S_3[t,i] \rightarrow S_2[t+1,i+1]: 1 \leq t < T-1 \text{ and } 1 \leq i < N-2\}
  \]
  \[
  R_{e_9} = \{T,N\} - \{S_3[t,i] \rightarrow S_2[t+1,i]: 1 \leq t < T-1 \text{ and } 1 \leq i < N-1\}
  \]
  \[
  R_{e_{10}} = \{T,N\} - \{S_3[t,i] \rightarrow S_2[t+1,i-1]: 1 \leq t < T-1 \text{ and } 2 \leq i < N-1\}
  \]
Domain and image of a relation associated with an edge $e_i$ represent the domain associated with the source and target vertices of $e_i$. For example, the domain and image of $R_{e_2}$ are as follows:

$$\text{domain}(R_{e_2}) = [N] \rightarrow \{S1[i]: 0 <= i < N\}$$

$$\text{image}(R_{e_2}) = [T,N] \rightarrow \{S2[t,i]: 1 <= t < T \text{ and } 1 <= i < N-1\}.$$ 

**Paths:** Given a path $P = (e_1, \ldots, e_l)$ in $G_F$ with associated edge relations $(R_{e_1}, \ldots, R_{e_l})$, the relation associated with $P$ can be computed by composing the relations of its edges, i.e.,

$$R_P = R_{e_l} \circ \cdots \circ R_{e_1}.$$ 

For instance, the relation for the path $(e_7, e_8)$ in the example, obtained through the composition $R_{e_8} \circ R_{e_7}$, is given by $R_P = [T,N] \rightarrow \{S2[t,i] \rightarrow S2[t+1,i+1]\}$. Further, the domain and image of a composition are restricted to the points for which the composition can apply, i.e.,

$$\text{domain}(R_{e_j} \circ R_{e_i}) = R_{e_i}^{-1}(\text{image}(R_{e_i}) \cap \text{domain}(R_{e_j}))$$

$$\text{image}(R_{e_j} \circ R_{e_i}) = R_{e_j}(\text{image}(R_{e_i}) \cap \text{domain}(R_{e_j})).$$

Hence, $\text{domain}(R_P) = [T,N] \rightarrow \{S2[t,i]: 1 <= t < T-1 \text{ and } 1 <= i < N-2\}$ and $\text{image}(R_P) = [T,N] \rightarrow \{S2[t,i]: 2 <= t < T \text{ and } 2 <= i < N-1\}$.

Two kinds of paths, namely, *injective circuit* and *broadcast path*, defined below, are of specific importance to our analysis.

**Definition 30** (Injective edge and circuit). An injective edge $e \in E_F$ is an edge of a flow-dependence graph $G_F = (V_F, E_F)$ whose associated relation $R_e$ is both affine and injective, i.e., $R_e = A_e \cdot x_e + b_e$, where $A_e$ is an invertible matrix. An injective
circuit is a circuit $P$ of a flow-dependence graph such that every edge $p \in P$ is an injective edge and its associated relation $\mathcal{R}_P = x_P + b_P$, where $I$ is an identity matrix. The vector $b_P$ is addressed as the dependence vector of the path $P$.

Note that it is sufficient for the matrix $A_e$ of the affine relation $\mathcal{R}_e$ of an injective edge $e$ to be any invertible matrix, however, the matrix of the composed affine relation of an injective circuit $P$ has to be an identity matrix.

**Definition 31** (Broadcast edge and path). A broadcast edge $b \in E_F$ is an edge of a flow-dependence graph $G_F = (V_F, E_F)$ whose associated relation $\mathcal{R}_b$ is affine and $\dim(\text{domain}(\mathcal{R}_b)) < \dim(\text{image}(\mathcal{R}_b))$. A broadcast path is a path $(e_1, \ldots, e_n)$ in a flow-dependence graph such that $e_1$ is a broadcast edge and $e_2, \ldots, e_n$ are injective edges.

Injective circuits and broadcast paths in a flow-dependence graph essentially indicate multiple uses of same data, and therefore are good candidates for lower bound analysis. Hence, only paths of these two kinds are considered in the analysis. The current example of Jacobi 1D computation illustrates the use of injective circuits to derive data movement lower bounds, while the use of broadcast paths for lower bound analysis is explained in another example that follows.

In the Jacobi example, we have three circuits between vertices $S2$ and $S3$. The relation for each circuit is computed by composing the relations of its edges as explained earlier. The relations, and the dependence vectors they represent, are listed below.
• Circuit $K_1 = (e_7, e_8)$:
  \[ R_{K_1} = [T, N] \rightarrow \{S_2[t,i] \rightarrow S_2[t+1,i+1] \mid 1 \leq t < T-1 \text{ and } 1 \leq i < N-2\} \]
  \[ b_1 = (1, 1)^T \]

• Circuit $K_2 = (e_7, e_9)$:
  \[ R_{K_2} = [T, N] \rightarrow \{S_2[t,i] \rightarrow S_2[t+1,i] \mid 1 \leq t < T-1 \text{ and } 1 \leq i < N-1\} \]
  \[ b_2 = (1, 0)^T \]

• Circuit $K_3 = (e_7, e_{10})$:
  \[ R_{K_3} = [T, N] \rightarrow \{S_2[t,i] \rightarrow S_2[t+1,i-1] \mid 1 \leq t < T-1 \text{ and } 2 \leq i < N-1\} \]
  \[ b_3 = (1, -1)^T \]

**Remark 2.** Each injective circuit in a flow-dependence graph gives rise to a set of vertex-disjoint directed paths in the corresponding CDAG. This is a direct consequence of injective property of the relations of injective circuits.

This fact is exemplified in Figure 6.5.

![Diagram of disjoint paths](image)

**Figure 6.5:** Set of disjoint paths in the CDAG corresponding to the injective circuit $K_1$. 

102
Frontiers: As in Figure 6.5, the sub-CDAG corresponding to injective circuits of a flow-dependence graph might have no input vertices. As discussed earlier, this might provide trivial bounds using 2S-partitioning technique. Hence, predecessor-free vertices have to be tagged as inputs (using Theorem 3) to obtain meaningful bounds. This set of predecessor-free vertices, called frontier of an injective circuit, can be obtained using the following set operation from a flow-dependence graph: A frontier, $F$, of an injective circuit $K$, with associated relation $\mathcal{R}_K$, can be calculated using the set operation $F = \text{domain}(\mathcal{R}_K) \setminus \text{image}(\mathcal{R}_K)$. The frontiers $F_1$, $F_2$ and $F_3$ of the three circuits, $K_1$, $K_2$ and $K_3$ respectively, are listed below.

- $F_1 = [T,N] \rightarrow \{ S2[1,i] : 2 \leq i < N-2 ; S2[t,1] : 1 \leq t < T-1 \}$
- $F_2 = [T,N] \rightarrow \{ S2[1,i] : 1 \leq i < N-1 \}$
- $F_3 = [T,N] \rightarrow \{ S2[1,i] : 2 \leq i < N-1 ; S2[t,N-2] : 2 \leq t < T-1 \}$

Injective circuits and lower bounds: Let $U \subseteq V$ be a vertex-set of a $2S_{NR}$-partition of the CDAG $C = (I,V,E,O)$ of Jacobi 1D computation. Corresponding to $U$, there are a set of points in the domain $\mathbb{Z}$-polyhedron $Z$ (e.g., the set of points inside the gray colored box in Figure 6.6). For simplicity, we just refer to the vertices in $C$ by their corresponding points in $Z$. As described in Remark 2, each injective circuit $K_i$, $1 \leq i \leq 3$, provides set of vertex-disjoint paths $P_i$ $1 \leq i \leq 3$, in $C$. In $Z$, the points outside the set $U$ with an edge to a point in $U$ corresponds to $\text{ln}(U)$. Since there is no cyclic dependence between the vertex-sets of a $2S_{NR}$-partition and the paths in set $P_i$, $1 \leq i \leq 3$, are vertex-disjoint, by starting from the vertices of $\text{ln}(U)$ and tracing backwards along any path $P_i$, $1 \leq i \leq 3$, we would reach $|\text{ln}(U)| \leq 2S$ distinct frontier vertices. This process of tracing backwards is equivalent
Figure 6.6: Iteration domain space for Jacobi 1D. Blue and red circles: Points representing dynamic instances of statements $S_2$ and $S_3$, respectively; Black arrows: Relation $\mathcal{R}_{K1}$ of injective circuit $(e_7, e_8)$; Black diamonds: Frontier $F_1$; Gray box: subset of points and corresponding vertex-set $U$.

to projecting the set $U$ along each dependence direction $b_i$, $1 \leq i \leq 3$ onto the frontier $F_i$, $1 \leq i \leq 3$. Hence, we have $|U_{\downarrow b_i}| \leq 2S$ (here, $V_{\downarrow b_i}$ denotes projecting $V$ along the direction $b_i$). We ensure that the selected set $U$ is inside the sub-CDAG under consideration by making sure that $U \subseteq D_U$, where $D_U = \text{domain}(\mathcal{R}_{K1}) \cap \text{image}(\mathcal{R}_{K1}) \cap \text{domain}(\mathcal{R}_{K2}) \cap \text{image}(\mathcal{R}_{K2}) \cap \text{domain}(\mathcal{R}_{K3}) \cap \text{image}(\mathcal{R}_{K3})$. Hence, injective circuits of a flow-dependence graph essentially provide us linear maps for valid projections required for application of geometric reasoning for lower bounds as discussed in Section 6.2.

Since, in our example, $\text{dim}(D_{S2}) = 2$, it is sufficient to consider any two of the three linearly independent dependence directions $b_i$, $1 \leq i \leq 3$ as projection directions.

Theorem 16 is applicable only for cases of orthogonal projections. If the projection directions are not parallel to the canonical directions, as in our example, a simple change of basis operation is sufficient to transform it to a new space where the projection directions are along the directions of the canonical bases. In the example,
if we consider vectors $\mathbf{b}_1/|\mathbf{b}_1|$ and $\mathbf{b}_3/|\mathbf{b}_3|$ as the projection directions in the original space, then the linear map $\left( \mathbf{b}_1/|\mathbf{b}_1| \quad \mathbf{b}_3/|\mathbf{b}_3| \right)^{-1}$ will transform the $\mathbb{Z}$-polyhedron to a new space where the projection directions are the canonical bases. In the example, after such transformation, the projection vectors are $(1, 0)^T$ and $(0, 1)^T$, and hence we have the following two linear maps: $\phi_1: (i, j) \rightarrow (i); \phi_2: (i, j) \rightarrow (j)$. We obtain the following inequalities for the dual problem (6.6): $x_1 \leq 1; x_2 \leq 1$. This process is illustrated in Figure 6.7.

![Figure 6.7: Original and transformed spaces for Jacobi 1D example.](a) Original space (a) Transformed space)

In addition, we also include constraints for the degenerate cases where the problem size considered may be small relative to the cache size, $S$. Hence, we have the following additional constraints for the example: $|\phi_1(U)| \leq (N + T) = S^{\log_S(N+T)}; |\phi_2(U)| \leq (N + T) = S^{\log_S(N+T)}$, or, $\sum_{i=1}^d x_i \delta_{i,1} \leq \log_S(N + T); \sum_{i=1}^d x_i \delta_{i,2} \leq \log_S(N + T)$. Hence, we obtain the constraints $x_1 \leq \log_S(N + T)$ and $x_2 \leq \log_S(N + T)$. Thus, we
solve the following following parametric linear programming problem.

Maximize $\Theta = x_1 + x_2$

s.t. $x_1 \leq \min(1, \log_S(N + T))$  \hspace{1cm} (6.8)

$x_2 \leq \min(1, \log_S(N + T))$

Solving (6.8) using PIP [21] provides the following solution:

If $\log_S(N + T) \geq 1$ then, $x_1 = x_2 = 1$, else, $x_1 = x_2 = \log_S(N + T)$, i.e., when $N + T = \Omega(S)$, $|U| = O(S^2)$, else, $|U| = O((N + T)^2)$. Hence, the data movement complexity of Jacobi 1D computation $Q$ satisfies

$$Q = \begin{cases} 
\Omega \left( \frac{NT}{S} - (N + T) \right), & \text{if } N + T = \Omega(S) \\
\Omega \left( \frac{NTS}{(N+T)^2} - (N + T) \right), & \text{otherwise}
\end{cases}$$

Here, $(N + T)$ is subtracted from the lower bound to account for input tagging.

In general, the change of basis operation that we performed earlier may not be unimodular, leading to a difference in the size of the set in the original and transformed spaces. Since we focus only on asymptotic parametric bounds, any constant multiplicative factors that arise due to the non-unimodular transformation are ignored.

### 6.3.2 Illustrative example 2

We use the example of matrix-matrix multiplication (matmult) to illustrate the use of broadcast paths for obtaining lower bounds. The code for matmult is presented in Listing 6.1 for reference.

**Domains and Relations:** The domain corresponding to the inputs and the statement $S1$ are listed below:

- $\mathcal{D}_A = [N] \rightarrow \{A[i,j] : 0 \leq i < N \text{ and } 0 \leq j < N\}$
Listing 6.1: Matrix-matrix multiplication.

- $D_B = [N] \rightarrow \{ B[i,j]: 0 \leq i < N \text{ and } 0 \leq j < N \}$

- $D_C = [N] \rightarrow \{ C[i,j]: 0 \leq i < N \text{ and } 0 \leq j < N \}$

- $D_{S1} = [N] \rightarrow \{ S1[i,j,k]: 0 \leq i < N \text{ and } 0 \leq j < N \text{ and } 0 \leq k < N \}$

Edges $e_1$, $e_2$ and $e_3$ represent the data dependence of statement $S1$ on the inputs $A$, $B$ and $C$, respectively. Edge $e_4$ represents the circuit from statement $S1$ to itself. The relations of the edges are as follows:

- $R_{e_1} = [N] \rightarrow \{ A[i,k] \rightarrow S1[i,j,k]: 0 \leq i < N \text{ and } 0 \leq j < N \text{ and } 0 \leq k < N \}$

- $R_{e_2} = [N] \rightarrow \{ B[k,j] \rightarrow S1[i,j,k]: 0 \leq i < N \text{ and } 0 \leq j < N \text{ and } 0 \leq k < N \}$

- $R_{e_3} = [N] \rightarrow \{ C[i,j] \rightarrow S1[i,j,0]: 0 \leq i < N \text{ and } 0 \leq j < N \}$

- $R_{e_4} = [N] \rightarrow \{ S1[i,j,k-1] \rightarrow S1[i,j,k]: 0 \leq i < N \text{ and } 0 \leq j < N \text{ and } 1 \leq k < N \}$

**Broadcast paths:** Let $e$ be any broadcast edge with associated relation $R_e$. It could be seen that once the points in $\text{domain}(R_e)$ are tagged as inputs, the set of points $\mathcal{I} \subseteq \text{domain}(R_e)$ obtained by projecting a set $\mathcal{Z} \subseteq \text{image}(R_e)$ onto $\text{domain}(R_e)$ using the inverse relation $R_e^{-1}$ provides us with $\ln(\mathcal{Z})$, i.e., $\mathcal{I} = \ln(\mathcal{Z}) \implies \mathcal{I} = R_e^{-1}(\mathcal{Z})$, $107$
where $e$ is a broadcast edge. Equivalently, by projecting a set $Z \subseteq \text{image}(R_e)$ along the direction of the kernel of $R_e$, $\ker(R_e)$, we obtain $\ln(Z)$. This property, in conjunction with Remark 2, allows us to use the kernel of the inverse relation of a broadcast path as a valid projection path for geometric reasoning. In our example, edges $e_1$ and $e_2$ are broadcast edges. Hence the paths $P_1 = (e_1)$ and $P_2 = (e_2)$ (consisting of just single edge each) are broadcast paths. Figure 6.8 shows the broadcast path $P_1$ and its corresponding sub-CDAG. We are specifically interested in inverse relations of the broadcast paths. Inverse relations of paths $P_1$ and $P_2$ are shown below.

$$
\mathcal{R}^{-1}_{P_1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
$$

$$
\mathcal{R}^{-1}_{P_2} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i \\ j \\ k \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}
$$

The kernels of relations $\mathcal{R}^{-1}_{P_1}$ and $\mathcal{R}^{-1}_{P_2}$ are $b_1 = (0, 1, 0)^T$ and $b_2 = (1, 0, 0)^T$, respectively. Further, we have the injective circuit $(e_4)$ with the dependence vector $b_3 = (0, 0, 1)^T$. Vectors $b_1$, $b_2$ and $b_3$ provide us three linearly independent projection directions that span the whole three-dimensional domain space.
Next, we calculate frontiers of the paths to find the set of points to be tagged as inputs. We calculate frontiers of broadcast paths using the same technique as before, by taking the set-difference of the domain and image of the relations. For example, frontier of the broadcast path $P_1$ is given by $F_1 = \text{domain}(R_{P_1}) \setminus R_{P_1}(\text{domain}(R_{P_1}))$.

The three frontiers corresponding to the three paths are listed below.

- $F_1 = [N] \rightarrow \{A[i,k] : 0\leq i < N \text{ and } 0\leq k < N\}$
- $F_2 = [N] \rightarrow \{B[k,j] : 0\leq k < N \text{ and } 0\leq j < N\}$
- $F_3 = [N] \rightarrow \{S1[i,j,0] : 0\leq i < N \text{ and } 0\leq j < N\}$

From the vectors $b_1, b_2$ and $b_3$, we obtain the following linear maps: $\phi_1 : (i,j,k) \rightarrow (i,k)$; $\phi_2 : (i,j,k) \rightarrow (j,k)$; $\phi_3 : (i,j,k) \rightarrow (i,j)$. This provides us the following inequalities: $x_1 + x_3 \leq 1$; $x_2 + x_3 \leq 1$; $x_1 + x_2 \leq 1$. Further, to handle degenerated cases, we have additional constraints that enforce the sizes of the projections onto the subspaces $\{i\}$, $\{j\}$ and $\{k\}$ to not exceed $N$, and sizes of the projections onto the subspaces $\{i,j\}$, $\{j,k\}$ and $\{i,k\}$ to not exceed $N^2$. Hence, we obtain the following parametric linear programming problem.

Maximize $\Theta = x_1 + x_2 + x_3$

s.t. \[ x_1 + x_2 \leq \min(1, 2 \log_S(N)) \]
\[ x_2 + x_3 \leq \min(1, 2 \log_S(N)) \]
\[ x_1 + x_3 \leq \min(1, 2 \log_S(N)) \] (6.9)
\[ x_1, x_2, x_3 \leq \log_S(N) \]

By solving Equation (6.9) using PIP [21] we obtain the following solution:

If $2 \log_S(N) \geq 1$ then, $x_1 = x_2 = x_3 = 1/2$, else, $x_1 = x_2 = x_3 = \log_S(N)$. Hence,
the data movement complexity of matmult $Q$ satisfies

$$Q = \begin{cases} 
\Omega \left( \frac{N^3}{\sqrt{S}} - N^2 \right), & \text{if } N = \Omega(\sqrt{S}) \\
\Omega(1), & \text{otherwise}
\end{cases}$$

### 6.3.3 Putting it all together

Algorithm 6.1 provides a pseudo-code for our algorithm. Since the number of possible paths in a graph is highly combinatorial, several choices are made to limit the overall practical complexity of the algorithm. First, only edges of interest, i.e., those that correspond to relations whose image is representative of the iteration domain, are kept. Second, paths are considered in the order of decreasing expected profitability. One criterion detailed here corresponds to favoring injective circuits over broadcast paths with one-dimensional kernel (to reduce the potential span), and then broadcast paths with decreasing kernel dimension (higher the kernel, more the reuse, lower the constraint). For a given vertex $v$, once the directions associated with the set of paths chosen so far span the complete space of the domain of $v$, no more paths are considered.

**try():** The role of the function try() (on lines 19, 24 and 29 in Algorithm 6.1) amounts to finding a set of paths that are linearly independent, compatible (i.e., a base can be associated to them), and representative. The function try() is shown in Algorithm 6.2.

**best():** The function best($v$) (shown in Algorithm 6.3) selects a set of paths for a vertex $v$ and computes the associated complexity. best($v$) is called only if the directions of the set of paths chosen for $v$ do not span the whole space.
solve(): The function solve() (shown in Algorithm 6.4) writes the parametric linear program and returns the data movement lower bound (with cases) for a domain \( D \) and a set of compatible subspaces.

Various operations used in the pseudo-code are detailed below.

- Given a relation \( R \), \( \text{domain}(R) \) and \( \text{image}(R) \) return the domain and image of \( R \), respectively.

- For an edge \( e \), the operation \( \text{relation}(e) \) provides its associated relation. If \( R_e = \text{relation}(e) \) has acceptable number of disjunctions, then the edge can be split into multiple edges with count equal to the number of disjunctions, otherwise, a convex under-approximation can be done.

- For a given path \( p = (e_1, e_2, \ldots, e_l) \) with associated relations \( (R_{e_1}, R_{e_2}, \ldots, R_{e_l}) \) we can compute the associated relation for \( p \) by composing the relations of its edges, i.e., \( \text{relation}(p) \) computes \( R_{e_l} \circ \cdots \circ R_{e_2} \circ R_{e_1} \). Note that the domain of the composition of two relations is restricted to the points for which the composition can apply, i.e. \( \text{domain}(R_i \circ R_j) = R_j^{-1}(\text{image}(R_j) \cap \text{domain}(R_i)) \) and \( \text{image}(R_i \circ R_j) = R_i(\text{image}(R_j) \cap \text{domain}(R_i)) \).

- For a given domain \( D \), \( \text{dim}(D) \) returns its dimension. If the cardinality of \( D \) (i.e., number of points in \( D \)) is represented in terms of the program parameters, its dimension can be obtained by setting the values of the parameters to a fixed big value (say \( B \)), and computing \( \log_B(|D|) \), and rounding the result to the nearest integer. For example, if \( |D| = C(n, m) = nm + n + 3 \), setting \( B = 10^3 \), we get \( \text{dim}(D) = \text{round} \left( \log_B \left( C(B, B) \right) \right) = 2 \).
Input: Flow-dependence graph $G_F = (V_F, E_F)$  
Output: Lower bound $Q$  
Data: $v$.complexity is an asymptotic complexity (with cases)  

Function $\text{FindLB}(G_F)$

1. $F_I \leftarrow \emptyset$;  
   // Set of injective edges
2. $F_B \leftarrow \emptyset$;  
   // Set of broadcast edges with 1D kernel
3. $F_{BB} \leftarrow \emptyset$;  
   // Set of broadcast edges with kernel dim $> 1$
4. foreach $e = (u, v) \in E_F$ do
   
   5. $R \leftarrow \text{relation}(e)$;
   6. if $\dim(\text{image}(R)) < \dim(D_v)$ then next $e$;  
      // $D_v$: Domain of $v$
   7. if $R$ is invertible then $F_I \leftarrow F \cup \{e\}$;
   8. if $\dim(\text{domain}(R)) = \dim(\text{image}(R)) - 1$ then
      9. $F_B \leftarrow F_B \cup \{e\}$
   10. if $\dim(\text{domain}(R)) < \dim(\text{image}(R)) - 1$ then
      11. $F_{BB} \leftarrow F_{BB} \cup \{e\}$

12. foreach $v \in V_F$ do
13.   $d \leftarrow \dim(D_v)$;
14.   foreach circuit $p$ from $v$ to $v$ in $F_I$ do
15.     $R \leftarrow \text{relation}(p)$;
16.     if $(b \leftarrow \text{ray}(R)) = \perp$ then next $p$;
17.     if $\dim(\text{image}(R)) < d$ then next $p$;
18.     if try$(v, \text{subspace}(b), p)$ then next $v$;
19.   foreach cycle-free path $p$ to $v$ in $F_{BB}$ do
20.     $R \leftarrow \text{relation}(p)$;
21.     if $(k \leftarrow \text{rk}(R)) = \perp$ then next $p$;
22.     if $\dim(\text{image}(R)) < d$ then next $p$;
23.     if try$(v, k, p)$ then next $v$;
24.   foreach cycle-free path $p$ to $v$ in $F_{BB}$ do
25.     $R \leftarrow \text{relation}(p)$;
26.     if $(k \leftarrow \text{rk}(R)) = \perp$ then next $p$;
27.     if $\dim(\text{image}(R)) < d$ then next $p$;
28.     if try$(v, k, p)$ then next $v$;
29.   best$(v)$;
30.   simplify$(\sum_{v \in V_F} v$.complexity$)$;

Algorithm 6.1: For each vertex $v$ in a flow-dependence graph $G_F$, finds a set of paths and computes the corresponding complexity.
Input: Vertex $v$; Subspace $k$; Path $p$

Data: $v.clique$ is a set of tuples $(b, K, D)$ where:
- $b$ is a basis,
- $K$ is a set of subspaces,
- $D$ is a domain,

Data: $v.complexity$ is an asymptotic complexity (with cases)

1 Function $try(v, k, p)$
2     foreach $(b, K, D) \in v.clique \cup \{ (\bot, \bot, \bot) \}$ do
3         $D' = \text{image}(\text{relation}(p)) \cap D$;
4         if $\dim(D') < \dim(D)$ then return false;
5         if $((b' \leftarrow \text{base}(b, k)) \neq \bot)$ then
6             $K' \leftarrow K \cup \{ k \}$;
7             $v.clique \leftarrow v.clique \cup \{ (b', K', D') \}$;
8             if $\text{dimension}(b') \geq \dim(\text{domain}(v))$ then
9                 $Q \leftarrow \text{solve}(D', K', b')$;
10                $v.complexity \leftarrow Q$;
11                return true;
12     return false;

Algorithm 6.2: For a vertex $v$, try to add path $p$ to some other paths. Return true if a good bound is found.

Input: Vertex $v$

1 Function $best(v)$
2     let $(b, K, D) \in v.clique$ with maximum lexicographic value of
3         $(\dim(D), \text{dimension}(b), -\sum_{k_i \in K} \text{dimension}(k_i))$;
4     $Q \leftarrow \text{solve}(D, K, b)$;
5     $v.complexity \leftarrow Q$;

Algorithm 6.3: For a vertex $v$, selects a set of paths and computes the associated complexity.

- If a relation $R$ is injective and can be expressed as an affine map of the form $I.x + b$, where $I$ is an identity matrix, then the operation $\text{ray}(R)$ computes $b$, otherwise, returns $\bot$. 
Input: Domain $D$; Set of subspaces $K$; Basis $b$

Function $\text{solve}(D, K, b)$
1. $b \leftarrow \text{base}(b, \ker(b));$  // Expand the basis to span full space
2. $LP \leftarrow \text{objective}(\text{maximize} \ \Theta = \sum_{i \in b} \alpha_i);$ 
3. foreach $k \in K$ do $LP \leftarrow LP, \text{constraint}(\sum_{i \notin k} b_i \alpha_i \leq 1);$ 
4. foreach $b' \subset b$ do $\quad$ // Degenerate cases
5. $\quad D_{b'} \leftarrow \text{projection}(\text{subspace}(b \setminus b', D));$
6. $\quad LP \leftarrow LP, \text{constraint}(\sum_{i \notin b'} b_i \alpha_i \leq \log_S(|D_{b'}|));$
7. $F \leftarrow \sum_{k \in K} |\text{projection}(-k, D)|;$  // Frontier
8. $\Theta \leftarrow \text{solution}(LP);$ 
9. $U \leftarrow S^\Theta;$ 
10. $Q \leftarrow \Omega \left(\frac{|D|S}{U} - F\right);$ 
11. return $Q;$

Algorithm 6.4: For a domain $D$ and a set of compatible subspaces, writes the linear program, and returns the data movement lower bound (with cases).

- For a relation $R$, if its inverse can be expressed as an affine relation $A \cdot x + b$, $\text{rkernel}(R)$ computes the kernel of the matrix $A$ (and returns $\bot$ otherwise).

- For a set of vectors $b = \{\vec{b}_1, \ldots, \vec{b}_l\}$, $\text{subspace}(b)$ provides the linear subspace spanned by those vectors.

- For a given basis $b'$ and a linear subspace $k$, $\text{base}(b', k)$ expands the basis $b'$ to include $k$, i.e., it gives a set of linearly independent vectors $b = \{\vec{b}_1, \ldots, \vec{b}_d\}$ such that $b' \subset b$ and for any $k_i$, there exists $b_i \subset b$ s.t. $k_i = \text{subspace}(b_i)$. If such a set could not be computed, it returns $\bot$.

- Given an expression $X$, the operation $\text{simplify}(X)$ simplifies the expression by eliminating the lower order terms. For example, $\text{simplify}(NT + N^2 - N + T)$ returns $NT + N^2$.  

114
• Given a set of points $P$ and a subspace $k$, $\text{projection}(k, P)$ returns the set of points obtained through orthogonal projection of $P$ onto the subspace $k$.

6.4 Related Work

Irony et al. [29] used Loomis-Whitney inequality to present lower bounds on the amount of data movement necessary for execution of standard matrix-matrix multiplication on a distributed-memory parallel system. They also showed a trade-off between local memory usage and the amount of communication that must be performed. Ballard et al. [4] generalized the technique in [29] and provided data movement lower bound results for various numerical linear algebra algorithms.

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
      C[i][j]+=A[i][k]*B[k][j];
```

**Listing 6.2:** Matrix-matrix multiplication

```
for(i=0;i<N;i++)
  for(j=0;j<N;j++)
    for(k=0;k<N;k++)
    {
      C[i][j] += 1;
      A[i][k] += 1;
      B[k][j] += 1;
    }
```

**Listing 6.3:** Code with same array accesses as matrix-matrix multiplication
As mentioned earlier, our approach in this chapter was inspired by the work of Christ et al. [15]. The two techniques differ in the following significant ways:

- The models of computation are different. Our work is based on the CDAG and pebbling formalism of Hong & Kung, while the lower bound results of Christ et al. [15] are based on the geometric model.

- The work of Christ et al. [15] does not model data dependencies between statement instances, and can therefore produce weak lower bounds. In contrast, the approach developed in this chapter is based on using precise data dependence information as the basis for geometric reasoning in the iteration space to derive the data movement lower bounds.

- Relation to data movement cost in our model is through the CDAG vertex-sets and their in-sets, while in [15], the relation is through statement instances and array access functions. For example, under their model, the lower bounds derived for codes in Listing 6.2 (standard matrix multiplication) and Listing 6.3 would be exactly the same – $\Omega(N^3/\sqrt{S})$ – since the analysis is based only on the array accesses in the computation. In contrast, with the red-blue pebble game model, the CDAGs for the two codes are very different, with the matrix-multiplication code in Listing 6.2 representing a connected CDAG, while the code in Listing 6.3 represents a CDAG with three disconnected parts corresponding to the three statements. The computation in Listing 6.3 has a much lower data movement complexity of $\Omega(N^2)$. In order to avoid obtaining such invalid bounds using the model in [15], a few additional assumptions have to
be made on the algorithm that is being analyzed and its implementations, as stated in [15, Theorem 4.1].
CHAPTER 7

Extension to Parallel Case

Multi-core/multi-node processors have become mainstream. These parallel computing environments come with hierarchical caches and interconnection networks with varying bandwidth at different levels. This has raised interest in extending the data movement complexity analysis to parallel systems, in order to determine the data movement bottlenecks between processors, and between different levels of memory within a processor. In this chapter, we develop an extension to $\text{RB}_{NR}$ pebble game model to capture the data movement requirements in large scale parallel systems. We consider multi-node multi-core parallel system, with total physical memory distributed across the nodes (connected through interconnection network). Section 7.2 presents the extended pebble-game model for parallel systems. In Section 7.3, we develop results required to characterize the intra-node and inter-node data movement costs. Interesting insights on architectural bottlenecks that limit the performance of algorithms are provided are provided in Section 7.4 using the techniques developed in this chapter.
7.1 Parallel Machine Model

Figure 7.1 shows our abstraction of a parallel computer. The model seeks to capture the essential characteristics of large-scale parallel systems, which exhibit multilevel parallelism and a hierarchical storage structure. The parallel computer has a set of multi-core nodes connected by an interconnection network. Each node has a number of cores and a hierarchy of storage elements: a set of private registers (at level 1) for each core, a private L1 cache per core (at level 2), and a hierarchy of zero or more additional levels of cache (through level $L - 1$), and a shared main memory (at level $L$). The total number of storage entities at level $l$ is denoted $N_l$, and the capacity of each entity at level $l$ is $S_l$ words. The model assumes that all caches at level $l$ have the same capacity of $S_l$ words. The hierarchical structure means that each storage entity at level $l$ is connected to a unique storage entity at level $l + 1$, and an integral multiple (usually a power of 2) of entities at level $l - 1$. The total number of main memory modules $N_L$ equals the number of nodes in the system. In summary, the following notations are used for our parallel machine model, for $1 \leq l \leq L$.

- $N_l$: Number of storage entities at level $l$.
- $c_{i,l}$: Storage entity $i$ at level $l$.
- $p_{i,l}$: Set of processors that share a storage entity $c_{i,l}$.
- $S_l$: Capacity of a single storage entity at level $l$.
- $N_l \times S_l$: Total storage capacity at level $l$.
- $\text{Pa}(c_{i,l})$: Function that returns the parent cache of $c_{i,l}$ in the memory hierarchy.
- $\text{Ch}(c_{i,l})$: Function that returns set of children caches of $c_{i,l}$ in the memory hierarchy.
7.2 Parallel Red-Blue Pebble Game

In this section, we present the framework for developing data movement lower bounds for parallel execution. In particular, we consider two types of data movement:

- Movement across the levels of the storage hierarchy within a node, called \textit{vertical data movement}

- Movement between nodes, called \textit{horizontal data movement}.

With the Parallel red-blue (P-RB$_{NR}$) pebble game, a different set of red pebbles is associated with each storage entity in the parallel system – we can consider there to be different shades of red, one per distinct storage entity. Associated with the storage entities at a level $l$ in the hierarchy, we have $N_l$ distinct shades of red pebbles, each associated with one of the $N_l$ distinct storage entities in the system at that level. The rules of the P-RB$_{NR}$ are stated below, and encode the constraints on movement of data in the parallel computer: i) vertical data movement can occur between physically connected entities in the storage hierarchy (Rules R4 and R5 below), and ii) data can
be moved via the interconnection network between the memories of any pair of nodes (Rule R3).

Let $C = (I, V, E, O)$ be a CDAG. Given for each level $l$, where $1 \leq l \leq L$, $N_l \times S_l$ number of red pebbles of different shades associated with different storage entities $c_{i,l}$, $1 \leq i \leq N_l$, and unlimited blue pebbles, the rules of the parallel red-blue ($\text{P-RB}_{NR}$) pebble game without recomputation are as follows:

**R1 (Input)** A level-$L$ red pebble can be placed on any vertex that has a blue pebble.

**R2 (Output)** A blue pebble can be placed on any vertex that has a level-$L$ shade red pebble on it.

**R3 (Remote get)** A level-$L$ red pebble can be placed on any vertex that has another level-$L$ red pebble.

**R4 (Move up)** For $1 \leq l < L$, a level-$l$ red pebble associated with $c_{i,l}$ can be placed on any vertex that has a level-$(l+1)$ red pebble associated with $c_{j,l+1}$, such that $1 \leq i \leq N_l$, $1 \leq j \leq N_{l+1}$ and $c_{i,l} = \text{Pa}(c_{j,l+1})$.

**R5 (Move down)** For $1 < l \leq L$, a level-$l$ red pebble associated with $c_{i,l}$ can be placed on any vertex that has a level-$(l-1)$ pebble associated with $c_{j,l-1}$, such that $1 \leq i \leq N_l$, $1 \leq j \leq N_{l-1}$ and $c_{i,l} \in \text{Ch}(c_{j,l-1})$.

**R6 (Compute)** If all predecessors of a vertex $v \in V \setminus I$ have a level-$1$ red pebble of shade $p$, and a level-$1$ red pebble of any shade has not been previously placed on $v$, then a level-$1$ red pebble may be placed on (or moved to) $v$. (Here $p$ denotes the index of the processor that computes vertex $v$.)

**R7 (Delete)** Any shade of red pebble may be removed from any vertex.
The goal of the game is to start with blue pebbles on all input vertices and end with blue pebbles on all output vertices using the rules of the game. Hence, any pebbling sequence for $P-RB_{NR}$ game contains a firing rule for each vertex in the operation set in the CDAG, where each vertex is fired at most once by any of the processors. The definitions of data movement cost, adapted for the parallel cost, are shown below.

**Definition 32** (Vertical data movement cost). For $1 < l \leq L$, the vertical data movement cost for a storage entity $c_{i,l}$, $vq_{i,l}(P, S_{l-1})$, of a pebbling sequence $P$ played using rules of $P-RB_{NR}$ pebble game with $N_{l-1} \times S_{l-1}$ level-$(l - 1)$ red pebbles on a CDAG $C$, is the sum of (1) the number of pebblings in $P$ according to the rule $R4$ applied on the vertices of $C$ holding red pebbles associated with $c_{i,l}$, and (2) the number of pebblings in $P$ according to the rule $R5$ applied on the vertices holding red pebbles associated with $c_{i,l-1}$ such that $c_{i,l-1} \in Ch(c_{i,l})$.

Informally, vertical data movement cost of a cache $c_{i,l}$ is the number of data transfers happening between $c_{i,l}$ and all its children.

**Definition 33** ($l$-optimal pebbling sequence). An $l$-optimal pebbling sequence for a CDAG is a pebbling sequence that minimizes the maximum data movement cost at level-$l$. In other words, if $vq_{i,l}(P, S_{l-1})$ denotes the cost of a pebbling sequence $P$ for a storage $c_{i,l}$, then an $l$-optimal pebbling sequence

$$P_{opt} = \arg\min_{\mathcal{P}} \max_{i : i \in [1, N_l]} vq_{i,l}(P, S_{l-1}).$$

**Definition 34** (Vertical level-$l$ data movement complexity). Vertical level-$l$ data movement complexity of a CDAG $C$, $Q_{l}(C, S_{l-1})$ is defined as the data movement cost of any $l$-optimal pebbling sequence.
The notation $Q_l(S_{l-1})$, or just $Q_l$ when $S_{l-1}$ is clear from the context, is used to denote the cost of any $l$-optimal pebbling sequence for a CDAG $C$.

**Definition 35** (Horizontal data movement cost of a pebbling sequence). The horizontal data movement cost for $c_{i,L}$, $hq_i(P, S_L)$, of a pebbling sequence $P$ played using rules of $P$-$RB_{NR}$ pebble game and with $N_L \times S_L$ level-$L$ red pebbles on a CDAG $C$, is the number of pebblings in $P$ according to the rule $R3$ applied on the vertices of $C$ resulting in a red pebble associated with $c_{i,L}$.

**Definition 36** ($H$-optimal pebbling sequence). A $h$-optimal pebbling sequence for a CDAG is a pebbling sequence that minimizes the maximum horizontal data movement cost.

**Definition 37** (Horizontal data movement complexity). Horizontal data movement complexity of a CDAG $C$, $Q_H(C, S_L)$ is defined as the data movement cost of any $H$-optimal pebbling sequence.

### 7.3 Lower Bounds for Parallel Data Movement Complexity

The hierarchical memory can enforce either the inclusion or exclusion policy. In case of inclusive hierarchical memory, when a copy of a value is present at a level-$l$, it is also maintained at all the levels $l + 1$ and higher. These values may or may not be consistent with the values held at the lower levels. The exclusive cache, on the other hand, does not guarantee that a value present in the cache at level-$l$ will be available at the higher levels. The following result is derived for the inclusive case. But, they also hold true for the exclusive case, where the difference lies only in the number of red pebbles that we consider in the corresponding two-level pebble game.
Theorem 17 (Vertical data movement lower bound). Given a CDAG $C = (I, V, E, O)$, and a level $l$, the vertical level-$l$ data movement complexity, $Q_l(S_{l-1})$, satisfies:

$$Q_l(S_{l-1}) \geq \frac{Q(N_{l-1} \times S_{l-1})}{N_l},$$

where $Q(S)$ denotes the serial data movement complexity of $C$ under $RB_{NR}$ pebble game model with $S$ red pebbles, $N_{l-1}$ is the total number of storage entities at level $l - 1$, and $S_{l-1}$ is the capacity of a single storage entity at level $l - 1$.

Proof. Consider an $l$-optimal pebbling sequence $P_{opt}$ of cost $Q_l(S_{l-1})$. Let $Q_l = \sum_{i=1}^{N_l} vq_{i,l}(P_{opt}, S_{l-1})$ denote the total vertical data movement cost of $P_{opt}$ at level-$l$. By Definition 33, we have, $N_l \times Q_l(S_{l-1}) \geq Q_l$. From $P_{opt}$, we can build a pebbling sequence $P$ for $RB_{NR}$ model of cost at most $Q_l$ with $S_{l-1} \times N_{l-1}$ red pebbles. $P$ is constructed by successively copying the pebblings from $P_{opt}$ to $P$ with the following changes: (1) Pebblings with rules R1, R2 and R3 are ignored, (2) pebblings with rule R4 applied on a vertex containing any level-$(l+1)$ red pebble are copied and changed as pebblings with rule R1 of $RB_{NR}$ game into $P$, (Note: Since we are assuming the caches to be inclusive, set of vertices with level-$k$, $k \leq l$, red pebbles are subset of vertices with level-$l$ red pebbles. Hence, there are at most $N_{l-1} \times S_{l-1}$ simultaneous vertices with level-$l$ and lower red pebbles.) (3) pebblings with rule R5 applied on a vertex containing any level-$l$ red pebble are copied and changed as pebblings with rule R2 into $P$, (4) pebblings with rule R6 are copied and changed as pebblings with rule R3 in $P$, and (5) pebblings with rule R7 involving level-$k$, $k \neq l$, red pebbles are ignored. Pebblings with rule R7 involving level-$l$ red pebbles are copied and changed as pebblings with rule R4 in $P$. 
\( P \) is a valid pebbling sequence for \( \text{RB}_{NR} \) pebble game. Rules R1, R2 and R4 are trivially satisfied in \( P \). When a vertex \( v \) is fired in \( P_{opt} \) all its predecessors have to have level-1 and higher, including level-\( l \), red pebbles. Hence all the predecessors of \( v \) in \( P \) would have a red pebble when \( v \) is fired in \( P \). Hence,

\[
Q_l(S_{l-1}) \geq \frac{Q_l}{N_l} \geq \frac{Q(N_{l-1} \times S_{l-1})}{N_l}.
\]

It is possible to obtain tighter lower bounds when using \( S \)-partitioning technique for deriving bounds. This is shown in the following theorem.

**Theorem 18.** Given a CDAG \( C = (I, V, E, O) \), and a level \( l \), the vertical level-\( l \) data movement complexity, \( Q_l(S_{l-1}) \), satisfies:

\[
Q_l(S_{l-1}) \geq \left( \frac{|V|}{|U| \times N_l} - \frac{N_{l-1}}{N_l} \right) \times S_{l-1}
\]

where \( U \) is the largest vertex set of \( C \) in any \( 2S_{l-1} \)-partition (under \( \text{RB}_{NR} \) model), \( N_{l-1} \) and \( N_l \) are the total number of storage entities at levels \( l-1 \) and \( l \), respectively, and \( S_{l-1} \) is the capacity of a single storage entity at level \( l-1 \).

The following theorem extends the \( S \)-partitioning technique to the horizontal case.

**Theorem 19.** Given a CDAG \( C = (I, V, E, O) \), the horizontal data movement complexity, \( Q_H(S_L) \), satisfies:

\[
Q_H(S_L) \geq \left( \frac{|V|}{|U| \times N_L} - 1 \right) \times S_L
\]

where \( U \) is the largest vertex set in any \( 2S_L \)-partition of \( C \), \( N_L \) is the total number of storage entities at level \( L \), and \( S_L \) is the capacity of a single storage entity at level \( L \).
7.4 Illustration of Use

Lower and upper bound analysis of algorithms can help us identify whether an algorithm is bandwidth bound at different levels of the memory hierarchy. The execution time, $T$, of an implementation on a computer is the maximum of its computational time and data movement time. Hence, if $T_{\text{comp}}$ denotes the computational time and $T_l$ denotes the data transfer time at level $l$, then, $T \geq \max(T_{\text{comp}}, T_l)$.

Computational time for an algorithm on a parallel computer system depends on the systems’ peak computational rate and the operation count of the algorithm. Let $C = (I, V, E, O)$ be the CDAG of an algorithm being analyzed. Then the operation count $W = |V \setminus I|$. If $F$ denotes the peak computational rate of a single processor, i.e., maximum number of floating-point arithmetic operations per second per core, then $T_{\text{comp}} \geq W/(P \times F)$, where $P$ is the total number of processors. Similarly, the data movement cost of an algorithm depends on the memory bandwidth and the data movement complexity of the algorithm. Hence, if $Q_l$ denotes the data movement complexity at a level $l$ and $B_l$ is the aggregate memory bandwidth between any level-$l$ storage entity and its children, then the data movement time at level-$l$, $T_l \geq Q_l/B_l$.

For the algorithm to be not bound by memory bandwidth at level $l$, its data movement time at level $l$ has to be less than the computational time of the algorithm. Hence, we need to have

$$\frac{Q_l}{B_l} \leq \frac{W}{P \times F} = \frac{W}{P_l \times N_l \times F},$$

where, $P_l$ is the number of processors that share a storage entity at level-$l$. By rewriting the above equation, we have the following equation that relates the machine
balance of a system with the data movement complexity of an algorithm.

\[ \frac{Q_l \times N_l}{W} \leq \frac{B_l}{P_l \times F}. \]

The right-hand side of the above equation represents the machine balance parameter of the system. If \( \text{LB}_l \) denotes the lower bound on the level-\( l \) data movement complexity \( Q_l \), then any algorithm that fails to satisfy the below equation will be invariably bandwidth bound at level-\( l \).

\[ \frac{\text{LB}_l \times N_l}{W} \leq \frac{B_l}{P_l \times F}. \quad (7.1) \]

Similarly, if \( \text{UB}_l \) is the upper bound on the level-\( l \) data movement complexity for an algorithm, then we can show that if the algorithm is bandwidth bound, then it would definitely satisfy the condition,

\[ \frac{\text{UB}_l \times N_l}{W} \geq \frac{B_l}{P_l \times F}. \quad (7.2) \]

Hence, if an algorithm fails to satisfy Equation (7.2), we can safely conclude that there is at least one execution order of \( C \) that is not constrained by the memory bandwidth at level \( l \).

Particularly, we are interested in understanding the memory bandwidth requirements (1) between the main memory and last level cache (LLC) within each node, and, (2) between different nodes, for various algorithms. For simplicity, we assume that the LLC is shared by all the cores within a node, which is common in practice.

Considering the particular case of data movement between LLC and the main memory, Equation (7.1) becomes,

\[ \frac{\text{LB}_L \times N_L}{W} \leq \frac{B_L}{P_L \times F}. \quad (7.3) \]
where, $P_L = P/N_L$. Similarly, for the inter-node data transfer, we have, Equation (7.2) becomes,

$$\frac{UB_H \times N_L}{W} \geq \frac{B_H}{N_L \times F}$$

(7.4)

where, $UB_H$ and $B_H$ represent the upper bound on the horizontal data movement complexity and the inter-node communication bandwidth, respectively.

Specifications for some of the computing systems are shown in table 7.1.

<table>
<thead>
<tr>
<th>Machine</th>
<th>$N_L$</th>
<th>Mem. (GB)</th>
<th>LLC (MB)</th>
<th>Vertical balance (word/s/FLOP)</th>
<th>Horizontal balance (word/s/FLOP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM BG/Q</td>
<td>2048</td>
<td>16</td>
<td>32</td>
<td>0.052</td>
<td>0.006</td>
</tr>
<tr>
<td>Cray XT5</td>
<td>9408</td>
<td>16</td>
<td>6</td>
<td>0.0256</td>
<td>0.005</td>
</tr>
</tbody>
</table>

Table 7.1: Specifications of computing systems.

7.4.1 Conjugate Gradient (CG)

**Vertical lower bound:** The pseudo-code for CG was provided in Algorithm 3.1. From Subsection 3.3.4, we obtained the data movement complexity of CG for serial case, $Q \geq 6n^dT$, where $n$ is the problem size. By applying Theorem 17, we obtain the vertical data movement lower bound for the data transfer between LLC and DRAM, $Q_L \geq 6n^dT/N_L$.

**Horizontal upper bound:** Here, we derive the data movement cost for an implementation of CG where the input problem grid is block partitioned among processors and each processor performs computation on the sub-grid assigned to it. Consider
the block partitioning of the input grid among the processors, with block size along each dimension $B = n/N_L^{1/d}$. Each processor holds the input data corresponding to its local grid points and computes the data needed by those grid points. Computation of the sparse matrix-vector product, at Line 5 in Algorithm 3.1, requires send and receive of values at the ghost cells of size $(B + 2)^d - B^d$ at each time-step with the neighboring processors. Hence, the horizontal data movement complexity, $Q_H$, satisfies,

$$Q_H \leq 2 \times ((B + 2)^d - B^d) \times T$$

$$= 2 \times (B^d + \binom{d}{1} B^{d-1}2^1 + \binom{d}{2} B^{d-2}2^2 + \cdots + \binom{d}{d-1} B^{1}2^{d-1} + \binom{d}{d} B^02^d - B^d) \times T$$

$$\leq 4dB^{d-1}T.$$  

**Analysis:** Equations (7.3) and (7.4) provided us conditions to determine the vertical and horizontal memory constraints of the algorithms. We will use them to show that the running time of CG is mainly constrained by the vertical data movement. We consider a three-dimensional problem ($d = 3$) for the analysis.

The vector dot-product at Line 7 requires $2n^d$ operations. The computation of $(r.r)$, at lines 10, 12 and 6, requires a single vector dot-product of operation count $2n^d$. The vector update operations at lines 8, 9 and 11 have operation count of $2n^d$ each. The SpMV operation at line 5 is a stencil computation with single time-step that requires $2.(2d + 1).n^d$ operations. This provides a total operation count of $24n^3T$ for a three-dimensional problem. The vertical I/O lower bound per node,
\( LB_L = 6n^3T/N_L \). Hence,

\[
\frac{LB_L \times N_L}{W} = \frac{(6n^3T/N_L) \times N_L}{24n^3T} = \frac{6}{24} = 0.25
\]

This value is higher than the vertical machine balance value of various machines (refer Table 7.1), leaving Inequality (7.3) unsatisfied. This shows that CG will be unavoidably bandwidth bound along the vertical direction for the problems that cannot fit into the cache. The only way to improve the performance would be to increase the main memory bandwidth.

On the other hand, let us consider the horizontal case.

\[
\frac{UB_H \times N_L}{W} = \frac{12B^2T \times N_L}{24n^3T}
\]

\[
= \frac{12 \times \left(n/N_L^{(1/3)}\right)^2 \times N_L}{24n^3} = \frac{\sqrt[3]{N_L}}{2n}
\]

This value is much lower than the horizontal machine balance values of various machines for the problem size typically encountered in practice, indicating that it is the intra-node memory bandwidth that is much more of a fundamental bottleneck than inter-node communication for CG.

7.4.2 Jacobi computation

**Vertical lower bound:** In Subsection 3.3.3, we derived a min-cut based serial lower bound for two-dimensional Jacobi computation. As shown in Section 7.3, it is possible to derive tighter parallel lower bounds using \( S \)-partitioning based technique. We derive an \( S \)-partitioning based parallel data movement lower bound for a \( d \)-dimensional Jacobi computation in this subsection. The proof of Theorem 20 below is closely related to the *information speed function* introduced in [27, Section 5].

130
Theorem 20 (S-partitioning based lower bound for $d$-dimensional Jacobi computation). For the $d$-dimensional Jacobi computation of size $n^d$ with $T$ time steps, the data movement complexity, $Q$, under two-level $RB_{NR}$ pebble game model satisfies $Q \geq \frac{n^d T}{4(2S)^{d/3}}$, where, $S$ is the number of red pebbles and $n \gg S$.

Proof. A $d$-dimensional Jacobi method performs computations on the points of a $d$ dimensional grid of size $n$ along each dimension, whose iteration domain is $d+1$ dimensions, containing $d$ space dimensions and one time dimension. Let $C = (I, V, E, O)$ be the CDAG corresponding to Jacobi computation. We will denote the vertex corresponding to the computation at a grid point $(p_1, \ldots, p_d)$ at time-step $t$ by $v(t, p_1, \ldots, p_d) \in V$. Jacobi computation has the property that computation of any point in the $d$-dimensional grid during a time-step $t$ requires the value computed for the same point at time-step $t-1$, i.e., we have edges $(v(t, p_1, \ldots, p_d), v(t+1, p_1, \ldots, p_d)) \in E$ for $1 \leq p_i \leq n$ and $1 \leq i \leq d$. These set of edges provide us $n^d$ vertex-disjoint paths $(v(1, p_1, \ldots, p_d), \ldots, v(T, p_1, \ldots, p_d))$ for $1 \leq p_i \leq n$ and $1 \leq i \leq d$. We will call these vertex-disjoint paths as lines for simplicity. Further, computation for a point depends on its nearest neighbors, i.e., each vertex $v(t, p_1, \ldots, p_d)$ depends on $v(t-1, p_1, p_2, \ldots, p_d), v(t-1, p_1, p_2, \ldots, p_d), \ldots, v(t-1, p_1, p_2, \ldots, p_d)$. Note that the vertices away from the boundaries depend on nearest neighbors along both $+$ and $-$ directions, while boundary vertices depend only on one direction. In either case, each vertex depends on at least one neighboring vertex along each dimension. Hence, with a path of length $m$, each vertex $v(t, p_1, \ldots, p_d)$ can be reached by vertices $v(t-m, p_1 \pm m, p_2 \ldots, p_d), v(t-m, p_1 \pm m, p_2 \ldots, p_d), \ldots, v(t-m, p_1, p_2, \ldots, p_d \pm m)$. Similarly, in the forward direction, each vertex $v(t, p_1, \ldots, p_d)$
can reach vertices \(v(t + m, p_1 \pm m, p_2 \ldots, p_d), v(t + m, p_1, p_2 \pm m, p_3, \ldots, p_d), \ldots, v(t + m, p_1, p_2, \ldots, p_d \pm m)\) using paths of length \(m\).

Consider any two vertices \(v_1 = v(t, p_1, \ldots, p_d)\) and \(v_2 = v(t + 2m, p_1, \ldots, p_d)\) in a same line at a distance \(2m\). With paths of length \(m\), \(v_1\) can reach the set of vertices \(W = \{v(t + m, p_1 \pm m, p_2 \ldots, p_d), v(t + m, p_1, p_2 \pm m, p_3, \ldots, p_d), \ldots, v(t + m, p_1, p_2, \ldots, p_d \pm m)\}\), where we have \(|W| \geq m^d\), and vertices in the set \(W\) can reach \(v_2\). Also, each vertex in the set \(W\) belongs to different lines. Hence, if a vertex-set \(E\) in a 2\(S\)-partition contains vertices \(v_1\) and \(v_2\), then it also necessarily contains vertices of \(W\), and since vertices in \(W\) are from at least \(m^d\) lines (i.e., vertex-disjoint paths), we have \(|\ln(E)| \geq m^d\). Choosing \(m \geq (2S)^{1/d}\) thus leads to a vertex-set with its in-set size at least \(2S\). Hence, any vertex-set of any \(2S_{NR}\)-partition will have vertices separated by a distance of at most \((2S)^{1/d}\) from each line. Hence, the largest vertex-set \(U\) in a \(2S_{NR}\)-partition can have at most \(2 \times (2S)^{1/d}\) vertices from each line. Also, \(U\) can only contain vertices from at most \(2S\) lines (since the lines are all vertex-disjoint). Hence, we have, \(|U| \leq 2 \times (2S)^{1/d} \times 2S = 2(2S)^{(d+1)/d}\). Hence, the data movement complexity \(Q\) satisfies \(Q \geq \left( \frac{n^d T}{2(2S)^{(d+1)/d}} - 1 \right) \times S\), which tends to \(Q \geq \frac{n^d T}{4(2S)^{1/d}}\) for \(n \gg S\). \(\square\)

**Theorem 21** (Vertical data movement complexity for \(d\)-dimensional Jacobi computation). For the \(d\)-dimensional Jacobi computation of size \(n^d\) with \(T\) time steps, the vertical level-\(l\) data movement complexity, \(Q_l\), satisfies \(Q_l \geq \frac{n^d T}{4N_l(2S_{l-1})^{1/d}}\), where, \(S_{l-1}\) is the capacity of a single storage entity at level \(l - 1\), \(N_l\) is the number of storage entities at level-\(l\) and \(n \gg S_{l-1}\).
Proof. In the proof for Theorem 20, we showed that the size of the largest vertex-set \( U \) of any \( 2S_{NR} \)-partition (under \( RB_{NR} \) model) of the CDAG for \( d \)-dimensional Jacobi computation satisfies \(|U| \leq 2(2S)^{(d+1)/d}\). Hence, for \( S = S_{l-1} \), from Theorem 18, the vertical level-\( l \) data movement complexity \( Q_l \) satisfies \( Q_l \geq \left( \frac{n^d T}{2N_l(2S_{l-1})^{(d+1)/d}} - \frac{N_{l-1}}{N_l} \right) \times S_{l-1} \), which tends to \( Q_l \geq \frac{n^d T}{4N_l(2S_{l-1})^{1/d}} \) for \( n \gg S_{l-1} \). □

**Horizontal upper bound:** The well-known distributed memory tiled Jacobi implementation incurs data communication cost at the boundaries due to the exchange of ghost cell values, similar to the Conjugate Gradient method. Hence, the upper bound on the horizontal data movement cost for the Jacobi method, \( Q_H \leq 4dB^{d-1}T \), where, \( B = n/N_{L}^{1/d} \) is the block size along each dimension.

**Analysis:** We consider a three-dimensional problem \((d = 3)\) for the analysis in this section. The operation count for a three-dimensional 7-point Jacobi method is \( 7n^3T \). The vertical data movement lower bound at the last level, \( LB_L = n^3T/(4\sqrt[3]{2S}N_L) \). Hence,

\[
\frac{LB_L \times N_L}{W} = \frac{\left( n^3T/(4\sqrt[3]{2S}N_L) \right) \times N_L}{7n^3T} = \frac{1}{35.3 \times \sqrt[3]{S}}
\]

For an LLC of size \( S_{l-1} = 6 \) MB (i.e., 0.75 MWords), \( LB_L = 3 \times 10^{-4} \) words/FLOP. This value falls below the vertical machine balance parameter of various architectures. Now, considering the vertical data movement upper bound per node, \( UB_L = 14n^d T/(S^{1/d}N_L) \).

\[
\frac{UB_L \times N_L}{W} = \frac{\left( 14n^3T/(\sqrt[3]{S}N_L) \right) \times N_L}{7n^3T} = \frac{2}{\sqrt[3]{S}}
\]

133
For $S = 0.75$ MWords, $UB_L = 0.022$ words/FLOP. This value falls slightly below the vertical machine balance parameters shown in Table 7.1, showing that the Jacobi method need not be necessarily bandwidth bound along the vertical direction, as opposed to CG.

Similarly, for the horizontal case,

$$\frac{UB_{\text{horiz}} \times N_L}{W} = \frac{12B^2T \times N_L}{7n^3T} = \frac{1.714 \times \sqrt[3]{N_L}}{n}$$

As in the case of CG, this value is much lower than the horizontal machine balance values of different machines for the common problem sizes, indicating that the horizontal data movement is not a bottleneck.

This shows that even though Jacobi method might require more iterations to converge, its lower ratio of I/O cost to the computational cost along both vertical and horizontal directions might make methods similar to it a more attractive alternative to CG for solving large sparse systems of linear equations on future machines that have more skewed machine balance parameters than current systems.

### 7.5 Related Work

Extending the scope of the Hong & Kung model to more complex memory hierarchies has been the subject of some research. Savage provided an extension together with results for some classes of computations that were considered by Hong & Kung, providing optimal lower bounds for data movement on a single processor machine with memory hierarchies [45]. Valiant proposed a hierarchical computational model [53] that offers the possibility to reason in an arbitrarily complex parameterized memory hierarchy model. The $P$-$RB_{NR}$ game developed in this dissertation extends the parallel model for shared-memory architectures by Savage and Zubair [47] to also include
the distributed-memory parallelism present in all scalable parallel architectures. The works of Irony et al. [29] and Ballard et al. [4] model communication across nodes of a distributed-memory system. Solomonik et al. [52] provide lower bounds on parallel execution of algorithms independent of number of processors, while deriving a trade-off between data movement, synchronization and computational costs. Scquizzato et al. [49] provided tight lower bounds on data movement complexity for computation of various algorithms on a distributed-memory parallel machines. Bilardi and Preperata [12] develop lower bound results for communication in a distributed-memory model specialized for multi-dimensional mesh topologies. Our model in this dissertation differs from the above efforts in defining a new integrated pebble game to model both horizontal communication across nodes in a parallel machine, as well as vertical data movement through a multi-level shared cache hierarchy within a multi-core node.
CHAPTER 8

Algorithm-Architecture Codesign Exploration

The roofline model is an insightful visual “bound and bottleneck” model that focuses on the performance limiting impact of off-chip memory bandwidth. In its most basic form, a performance roofline plot (explained in greater detail with Figure 8.1 in the Section 8.1.1) consists of two straight lines that represent upper bounds on performance due to the maximum computational rate of processor cores and memory bandwidth, respectively. The horizontal axis is the operational intensity (OI) of the computation, defined as the ratio of number of computational operations performed per byte of data moved between main memory and the processor. A code will be memory-bandwidth limited unless OI is sufficiently high, greater than a “critical intensity” corresponding to the point of intersection of the two rooflines.

Modeled or measured data volume between main-memory and last-level on-chip cache (LLC) for the execution of a code can be used to compute its OI and thus determine whether or not it is in the bandwidth-limited region of the roofline model for a machine. However, a significant issue is that the OI of an algorithm is not a fixed quantity, but depends on the on-chip LLC capacity and the problem size, and is affected by how the algorithm is implemented as well as the semantics-preserving code transformations that may be performed. Thus, it is only straightforward to use
a roofline model to determine whether a particular implementation of an algorithm
is bandwidth bound on a particular machine. But we are often interested in broader
insights on performance bottlenecks, where we wish to take into consideration po-
tential transformations into semantically equivalent forms, or consider architectural
variations in a design space. But this is not easy. The standard approach to doing
so requires performance modeling and analysis of a number of alternative implement-
tion scenarios.

In this chapter, we present a new approach to use of the roofline model by de-
riving upper bounds on $OI$ from lower bounds on data movement complexity. Since
data movement movement lower bounds are schedule-independent, its use avoids the
need to explicitly model and analyze $OI$ over a large number of mapping/scheduling
scenarios.

8.1 Background

8.1.1 The roofline model

The roofline model [55] is a popular approach to bottleneck-bounds analysis that
focuses on the critical importance of the memory bandwidth in limiting performance.

Performance roofline: The performance roofline sets an upper bound on perfor-
mane of an implementation depending on its operational intensity. Let $W$ be the
number of operations (typically floating-point computations) that a particular imple-
mentation of an algorithm performs and let $q$ be the total number of bytes it transfers
between main memory and the processor. (Table 8.1 summarizes the parameters.)

The operational intensity $(OI)$ of a computation is defined as the ratio be-
tween $W$ and $q$, i.e., $OI = W/q$. The performance roofline model ties together the
operational intensity, peak floating-point performance and memory performance in a two-dimensional graph. The horizontal axis (in log scale) represents the operational intensity and the vertical axis (in log scale) represents the attainable performance. The attainable performance for an architecture depends on its peak floating-point performance and the memory bandwidth, and is given by:

\[
\text{Upper bound on attainable performance} = \text{Min}(F_{\text{flops}}, B_{\text{mem}} \times OI).
\]
In its most basic form, a performance roofline model contains two intersecting straight lines representing performance limits: a horizontal line at a height corresponding to the peak computational performance of the core(s), and an inclined line with y-intercept corresponding to the peak memory-to-LLC bandwidth. This is shown in Figure 8.1(a).

The OI of a given implementation of an algorithm is typically computed using performance counters by measuring the actual number of operations and amount of memory traffic during execution. In some cases, it may be possible to calculate OI through manual reasoning. If we draw a vertical line (red and green lines in Figure 8.1(a)) at this value of OI, it intersects either the inclined line indicating that the implementation is bandwidth-bound, or the horizontal line indicating that the implementation is compute-bound. For lower OI, the memory bandwidth is a fundamental limiting factor, and achievable computational throughput cannot exceed the product of OI and the memory bandwidth. As OI increases from zero, the upper bound on achievable performance steadily increases, up to the point of intersection of the two lines. To the right of the intersection point, OI is sufficiently high that the memory bandwidth is no longer a fundamental constraint on achievable computational throughput.

**Energy roofline:** The roofline model has recently been adapted [13, 14] to capture upper bounds on energy efficiency, i.e., operations/joule, as a function of OI. For a particular OI, the energy cost of at least one memory access must be expended for every OI computational operations. The minimal energy cost for performing a set
of $W$ operations is the sum of the energy cost for actually performing the $W$ arithmetic operations and the energy cost for $W/OI$ memory operations. Figure 8.1(b) shows the energy roofline for the same architecture as the performance roofline (in Figure 8.1(a)). The energy roofline is smooth since the total energy is the sum of compute and data movement energies, in contrast to execution time, which is bounded by the larger of the data movement time and compute time (since data movement and computation may be overlapped).

Given a particular algorithm, e.g., the standard $O(N^3)$ algorithm for matrix-matrix multiplication, there exist many semantically equivalent implementations of the algorithm with different values of $OI$. In general, finding the schedule with the highest possible $OI$ would require analysis of a combinatorially explosive number of schedules. In contrast, with the approach we develop in this chapter using the data movement lower bounds, a single parametric expression can be developed for an upper bound on $OI$ for a regular CDAG as a function of the size of LLC. The upper bound captures all possible valid schedules for the CDAG, including all tiled versions of the code, considering all possible tile sizes.

### 8.2 Operational Intensities of Algorithms

In this section, we develop expressions for upper bounds on $OI$ for four algorithms as a function of storage capacity – matrix multiplication (MM), Fast Fourier transform (FFT), conjugate gradient (CG) and 9-point 2D Jacobi computation (J2D). The $OI$ upper bounds are used in the following sections for architecture-algorithm co-design exploration. The upper bounds on $OI$ are obtained using data movement lower bounds. In modeling $OI$ in operations per byte, we assume the data to be double-
precision occupying 8 bytes per word and each red/blue pebble to have a capacity of one word.

**Matrix-Matrix Multiplication (MM):** The standard matrix-matrix multiplication algorithm has an operation count, $W_{MM} = 2N^3$ for the product of $N \times N$ matrices. Irony et al. [29] showed that the data movement complexity for MM $Q_{MM} \geq N^3/2\sqrt{2S}$, for $S$ red pebbles. Hence, the operational intensity $OI_{MM}$ satisfies,

$$OI_{MM} \leq \frac{2N^3}{N^3/(2\sqrt{2S})} = 4\sqrt{2S} \text{ FLOP/Word, or } 0.5\sqrt{2S} \text{ FLOP/Byte.}$$

**Fast Fourier Transform (FFT):** In Subsection 3.3.2, we developed a data movement lower bound for FFT, where we showed that the data movement complexity of an N-point FFT, $Q_{FFT} \geq 2N \log(N)/\log(S)$. Further, an N-point FFT has an operation count of $W_{FFT} = 2N \log(N)$. Hence, the operational intensity for FFT,

$$OI_{FFT} \leq \frac{2N \log(N)}{(2N \log(N))/\log(S))} = \log(S) \text{ FLOP/Word, or } 0.125 \log(S) \text{ FLOP/Byte.}$$

**Conjugate Gradient (CG):** The pseudo-code for CG was provided in Algorithm 3.1. For the analysis in this chapter, we consider a two-dimensional input problem being solved with CG. For a two-dimensional problem, operational count of CG is $W_{CG} = 20N^2T$. Further, from Subsection 3.3.4, we obtained the data movement complexity of CG, $Q_{CG} \geq 6N^2T$ for $d = 2$. Hence, we have, the operational intensity for CG,

$$OI_{CG} \leq 20/6 \text{ FLOP/Word, or } 0.417 \text{ FLOP/Byte.}$$
9-point Jacobi 2D (J2D): From Subsection 3.3.3, we have the data movement complexity for J2D, \( Q_{J2D} \geq \frac{0.75n^2T}{\sqrt{S}} \). The 9-point Jacobi has operation count of \( W_{J2D} = 9N^2T \). Hence, we obtain the operational intensity for J2D,

\[
OI_{J2D} \leq 12\sqrt{S} \text{ FLOP/Word, or } 1.5\sqrt{S} \text{ FLOP/Byte.}
\]

8.3 Architecture Design Space Exploration

In the next two sections, we show how an analysis of upper bounds on \( OI \) of an algorithm as a function of cache size can be used to model the limits of achievable performance as well as energy efficiency. We perform two types of analysis:

- For a particular set of technology parameters, what are the upper bounds on achievable per-chip performance (GFLOP per second) for different algorithms, considering a range of possible ways of dividing the chip’s area among processor cores versus cache memory (Section 8.4)?

- For a given set of technology parameters, what are the upper bounds on energy efficiency (Giga-operations per Joule) for different algorithms (Section 8.5)?

In this section, we provide details on how various architectural parameters were chosen for the analysis.

8.3.1 Notations

We consider a chip with a fixed total area \( A \), where a portion \( \alpha A \) of the total area is used for the last level cache and the remaining area, \( (1 - \alpha)A \), is allocated to cores. While such an analysis could be directly extended to model constraints with respect to multiple levels of cache, we only model the impact of the last level
of cache (LLC) in this analysis. Also, we do not account for the area required for interconnects and other needed logic. It is important to note that since our modeling is purely an upper bound analysis on performance and energy efficiency, the results obtained are of course valid under these simplifications: including more details of the architecture can allow tighter bounds, but all results presented later in this section are assertable upper limits on performance and energy efficiency for the modeled technology. For example, when the analysis shows that large-FFT (size too large to fit in LLC) performance cannot exceed 60 GFLOPs aggregate performance for a multi-core chip built using the modeled technology, it represents an upper bound that cannot be exceeded by any possible actual implementation with those technology parameters. A more detailed model that accounts for additional levels of cache, on-chip interconnect, control logic, etc., may serve to tighten or reduce the upper bound to a lower value than 60 GFLOPs, but cannot invalidate the conclusions drawn from the more simplified analysis.

The following valid simplifications are made to make the analysis easier:

- Un-core and I/O clock frequencies are assumed to be fixed.

- Interconnects are not considered in the analysis.

- LLC is considered to be shared by all the cores.

In our architectural design space exploration, the number of cores, $P$, and the total number of locations in the LLC, $S$, are directly related to the total die area $A$ and the parameter $\alpha$ trades off one for the other. $F_{\text{flops}}(f)$ varies with the clock frequency $f$. 

143
Table 8.2: Notations used in architecture design space exploration.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>Total chip area</td>
</tr>
<tr>
<td>$\alpha A$</td>
<td>Area occupied by LLC</td>
</tr>
<tr>
<td>$(1 - \alpha)A$</td>
<td>Area occupied by cores</td>
</tr>
<tr>
<td>$f$</td>
<td>Clock frequency (GHz)</td>
</tr>
<tr>
<td>$P$</td>
<td>Number of cores</td>
</tr>
<tr>
<td>$S$</td>
<td>Last level cache size in words</td>
</tr>
<tr>
<td>$\pi_{cpu}$</td>
<td>Static leakage power per core</td>
</tr>
<tr>
<td>$\pi_{cache}$</td>
<td>Static leakage power for cache of size $S$</td>
</tr>
<tr>
<td>$\epsilon_{flop}$</td>
<td>Energy per arithmetic operation at frequency $f$</td>
</tr>
<tr>
<td>$\epsilon_{mem}$</td>
<td>Energy per byte of data access from/to memory</td>
</tr>
</tbody>
</table>

**Performance:** The total computation time $T$ follows the roofline model and can be expressed as:

$$T = \max \left( \frac{W}{P\cdot F_{flops}(f)/B_{mem}}, \frac{8Q}{B_{mem}} \right) = \max \left( \frac{W}{P\cdot F_{flops}(f)/B_{mem}}, \frac{W}{B_{mem}\cdot OI(S)} \right) \quad (8.1)$$

**Energy:** The total energy consumption is modeled as:

$$E = W\cdot \epsilon_{flop}(f) + 8Q\cdot \epsilon_{mem} + (P\cdot \pi_{cpu} + \pi_{cache}(S))\cdot T, \quad (8.2)$$

where the quantities used are defined in Table 8.2.

### 8.3.2 Architectural Parameters

We demonstrate the utility of modeling upper bounds on operational intensity by performing an analysis over possible architectural variants for a given technology. We use architectural parameters for an enterprise Intel Xeon processor (codename Nehalem-EX) [44] and use the published and available statistics to estimate area, power and energy for different compute and memory units.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die Size</td>
<td>684 mm²</td>
</tr>
<tr>
<td>Technology Node</td>
<td>45 nm</td>
</tr>
<tr>
<td>Num of Cores</td>
<td>8</td>
</tr>
<tr>
<td>Num of LLC Slices</td>
<td>8</td>
</tr>
<tr>
<td>LLC Slice Size</td>
<td>3MB</td>
</tr>
<tr>
<td>LLC Size (total)</td>
<td>24 MB</td>
</tr>
<tr>
<td>DRAM Channels</td>
<td>4</td>
</tr>
<tr>
<td>Core Voltage</td>
<td>0.85 – 1.1 V</td>
</tr>
<tr>
<td>Core Max Clock</td>
<td>2.26 GHz</td>
</tr>
<tr>
<td>TDP</td>
<td>130 W</td>
</tr>
<tr>
<td>Threads per core (SMT)</td>
<td>2</td>
</tr>
<tr>
<td>Leakage (total)</td>
<td>21 W</td>
</tr>
<tr>
<td>L1/L2</td>
<td>32KB / 256 KB</td>
</tr>
</tbody>
</table>

**Table 8.3:** Nehalem-EX processor spec.

Table 8.3 shows the physical specifications for a testbed CPU. Using the die dimensions, and by processing the die photo, we computed the area (in mm²) for the chip-level units of interest. For this study, we model core area, which includes private L1 and L2 caches. We also model a range of values for the shared last level cache (LLC).

<table>
<thead>
<tr>
<th>Unit</th>
<th>Width (mm)</th>
<th>Height (mm)</th>
<th>Area (mm²)</th>
<th>Area (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Die</td>
<td>31.9</td>
<td>21.4</td>
<td>684</td>
<td>100</td>
</tr>
<tr>
<td>Core</td>
<td>7.2</td>
<td>3.7</td>
<td>26.7</td>
<td>3.9</td>
</tr>
<tr>
<td>LLC Slice</td>
<td>7.8</td>
<td>3.7</td>
<td>29.3</td>
<td>4.3</td>
</tr>
</tbody>
</table>

**Table 8.4:** Nehalem-EX die dimensions.
Table 8.5: Modeled LLC Slices using CACTI [38]

Table 8.4 shows the extracted dimensions for the Nehalem-EX Core and LLC. In order to explore LLC with different sizes, we used CACTI [38]. We fixed LLC design parameters based on the chip specification, varying only the size of the cache.

Table 8.5 shows modeled cache sizes and their corresponding area (from CACTI). Using the area percentage for each unit, and the reported total leakage power for the chip (21W, or 16%), we modeled the static power consumed by each core on an area-proportional basis.

In order to model dynamic power per core, we used McPAT [35], an analytical tool to model the CPU pipeline and other structures. To estimate core parameters, we extended the available Xeon model to allow for a greater number cores. All modifications were based on real parameters published by Intel [44]. In order to model
the impact of voltage/frequency scaling on energy efficiency, we extracted the maximum and minimum operating voltage range for the processor, and the corresponding frequencies at which the processor can operate. Using those V/F pairs, different “power states” were modeled for the processor using McPAT [35]. Table 8.6 shows how changing voltage and frequency effects total chip and core power.

<table>
<thead>
<tr>
<th>Voltage (V)</th>
<th>Clock (MHz)</th>
<th>Peak Chip Dyn (W)</th>
<th>Peak Core Dyn (W)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.10</td>
<td>2260</td>
<td>111.14</td>
<td>102.68</td>
</tr>
<tr>
<td>1.05</td>
<td>2060</td>
<td>95.94</td>
<td>87.49</td>
</tr>
<tr>
<td>1.00</td>
<td>1860</td>
<td>81.82</td>
<td>73.35</td>
</tr>
<tr>
<td>0.95</td>
<td>1660</td>
<td>62.23</td>
<td>60.78</td>
</tr>
<tr>
<td>0.90</td>
<td>1460</td>
<td>58.12</td>
<td>49.66</td>
</tr>
<tr>
<td>0.85</td>
<td>1260</td>
<td>48.39</td>
<td>39.93</td>
</tr>
</tbody>
</table>

**Table 8.6:** Nehalem-EX: effect of changing voltage and frequency on core/chip power

Finally, we summarize the parameters for the testbed architecture used for the modeling in the next sections: $F_{flops}(f) = 9.04$ GFLOPS @ 2.26 GHz, $B_{mem} = 40$ GB/s, $A = 684$ mm$^2$, $A_{core} = 26.738$ mm$^2$, $\epsilon_{flop}(f) = 1.3$ nJ/flop, $\epsilon_{mem} = 0.63$ nJ/byte, $\pi_{cpu} = 0.819$ W. Each core, which includes private L1/L2 caches, occupies around 4% of the total die area. For a very small value of $\alpha$, a maximum of 25 cores would fit on the chip. At the other extreme, for $\alpha$ of 0.95, only one core can be put on the chip and an LLC cache of 64 MBytes could be accommodated.

### 8.4 Algorithm-Architecture Codesign: Bounds on Performance

In this section, for different algorithms, we consider the implications of upper bounds on $OI$ on the maximal performance (operations per second) achievable on
a chip. As described in the previous section, we assume that the chip area can be partitioned as desired among processor cores or cache. As shown below for four demonstration benchmarks, the upper bound on $OI$ for an algorithm can be modeled as a monotonic non-decreasing function of cache size. As the fractional chip area occupied by LLC increases, the upper bound on $OI$ increases. But simultaneously, the fractional area usable for cores decreases, so that fewer cores may be placed on the chip. As shown next, a collection of rooflines can be used to capture the architecture design space. Alternatively, we show that the collective data can be captured in a single multi-roofline plot.

The size of the LLC was varied from a tiny 4 Kbyte size (representing a fraction of under 0.1% of the chip of approximately 700 $mm^2$ area) to 64 Mbytes (filling almost the entire chip area). For demonstration purposes, we analyze four algorithms: matrix multiplication (MM), fast Fourier transform (FFT), conjugate gradient (CG) and 9-point 2D Jacobi (J2D) iterative linear system solvers. In all these cases, the problem sizes were assumed to be much larger than LLC, when $OI$ is essentially independent of the problem size, as per the analysis in Section 8.2. The following upper bounds were obtained for $OI$, in FLOP/Byte, as a function of cache capacity ($S$ words), for double-precision data occupying 8 bytes per word.:

- MM: $0.5\sqrt{2S}$
- FFT: $0.125 \log(S)$
- CG: 0.417 (it is independent of cache capacity)
- J2D: $1.5\sqrt{S}$
Figure 8.2 shows the variation of upper bounds on $OI$ as a function of fractional chip real estate used for cache, assuming double precision data occupying 8 bytes per word. It may be seen that the trends for MM and J2D are similar, while that of remaining two are very different. MM and J2D show a significant increase in $OI$ as the fraction of chip area occupied by cache is increased (the plot is logarithmic on the $y$-axis). FFT shows a very mild rise in $OI$ as the cache fraction is increased, and the $OI$ is low – between 0.5 and 2 over the entire range. CG has a flat and very low value of $OI$ (0.42), irrespective of the amount of cache deployed.

![Figure 8.2: Operational intensity upper bounds as a function of $\alpha$.](image)

Figure 8.3 shows four roofline plots for four specific values of $\alpha$: 0.01, 0.25, 0.5 and 0.95, respectively. In each case, the vertical lines are placed at the upper bound on $OI$ for the four algorithms, and either intersect the inclined bandwidth roofline or the horizontal processor-peak roofline.
Figure 8.3: Upper bounds on performance for different choices of \( \alpha \).

At a very small value of \( \alpha \) of 0.01, the size of LLC is very small and a maximal number of cores (25) can be put on the die. So the horizontal roofline is at a performance of 226 GFLOPs (9.04*25). The \( OI \) values of the four benchmarks – MM, FFT, CG and J2D with this configuration are 0.41, 2.0, 181.02 and 384.0, respectively. CG and FFT are bandwidth-bound, although not to the same extent, while MM and J2D are not bandwidth-bound.
When $\alpha$ is increased to 0.25, the number of cores that can be placed on chip decreases, causing a lowering of the horizontal roofline. The $OI$ values increase for FFT, MM and J2D, while they are unchanged for CG. Compared to $\alpha$ of 0.01, the performance upper bound for FFT increases because the intersection with the bandwidth roofline occurs further to the right. But for MM and J2D, the performance upper bounds decrease despite a higher $OI$, since the horizontal roofline has dropped due to fewer cores. It is interesting to note that the trends as a function of $\alpha$ are in opposite directions for FFT and MM.

![Figure 8.4: Upper bounds on performance.](image)

Figure 8.4 shows a single combined roofline plot that captures the variation of upper bounds on performance for the entire design space of configurations of the chip, i.e., over the range of possible values of $\alpha$. The plot represents a consolidated analysis that takes into account the inter-dependence between the size of LLC and the maximum number of cores — the larger the LLC, the less the remaining area on the
chip for cores. The value of $\alpha$ determines how many cores can be placed on the chip. With the parameters of the chosen technology detailed in the previous section, each core occupies a little under 4% of the chip area. The horizontal rooflines corresponding to four values of $\alpha$ are shown in the figure, intersecting with the bandwidth roofline (corresponding to 40 Gbytes/sec). Four instances of horizontal rooflines are shown in the figure, tagged with the value of $\alpha$ and corresponding number of cores.

The results for the three benchmarks – MM, FFT and CG – show disjoint ranges of $OI$. CG has no variation in $OI$ as a function of $S$. Its $OI$ of 0.417 leads to an upper bound of 16.7 GFLOPs ($0.417 \times 40 \text{Gbytes/sec}$). Since each core has a peak performance of 9.04 GFLOPs for the modeled technology, CG is bandwidth-bound for any number of cores greater than one. But if $\alpha$ is 95%, we can only put one core on the chip, and the upper bound is 9 GFLOPs. Thus, the two red dots in the multi-roofline plot capture the entire range of possibilities as alpha is varied: 9 GFLOPs when $P = 1$ ($\alpha$ is above 0.93) and 16.7 GFLOPs when $P \geq 2$.

For FFT, the upper bound on $OI$ ranges from 1.125 to 2.875 over the range of $\alpha$ from 0.1% to 95%. At $\alpha = 0.05\%$, $OI = 1.125$, and the performance upper bound is 45 GFLOPs, and the computation is severely bandwidth bound – the 25 cores that could be put on chip for this value of $\alpha$ would be heavily under-utilized. As $\alpha$ is increased, the upper bound on performance improves, and hugs the bandwidth roofline. FFT’s performance reaches its peak when $\alpha = 50\%$ and the number of cores is 13. But when $\alpha$ goes to or above 75%, and the number of cores drops below 6, the algorithm becomes compute bound because the peak computational performance is lower than $40 \times OI(S)$.  

152
MM has a range (as the size of the LLC is varied) that is always in the compute-bound region of the roofline. But as the LLC size is increased, the number of cores on chip must decrease, and the performance upper bound drops at very high values of $OI$.

Jacobi 2D follows the same trend at that of MM and is always in the compute-bound region. The performance of J2D for different values of $\alpha$ is exactly equal to that of MM, even though J2D has higher $OI$ than MM. Hence, MM and J2D perform the best when maximum number of cores of $P = 25$, for $\alpha \leq 1\%$, is present.

The analysis shows that two currently widely used algorithms, FFT and CG, will not be well suited for solution of very large problems relative to the cache capacity, unless the bandwidth between main memory and cache is substantially increased relative to representative parameters of current systems.

Analysis of lower bounds on data movement and upper bounds on $OI$ can be similarly carried out for any level of the storage hierarchy - including data movement between cache and registers, or disk and main memory. The bounds for any level can be obtained by appropriately setting the value of “$S$”. Here, we have focused on the data movement between the memory and on-chip LLC to aid in deriving bounds on performance for algorithm-architecture codesign because off-chip memory bandwidth is often a critical performance bottleneck. The presented methodology can however be similarly applied to identify potential bottlenecks at other levels of the cache hierarchy.

The lower bounds on data movement and the corresponding upper bounds on $OI$ are inherent fundamental limits that cannot be overcome. We note that although the above analysis has made several simplifying assumptions, the upper bounds on
performance can be asserted to hold even when any simplifying assumptions are removed and more accurate information is used. For example, if information about area occupied by interconnects is added, the total number of cores and the total LLC cache size will go down but not increase. Hence all observations on upper bounds on performance will still be valid. Interconnects may add latencies and also bandwidth constraints, but they will not invalidate any of the above conclusions on performance upper bounds for the modeled technology parameters. Similarly, although in reality full overlap of computation and communication is not feasible, and perfect scaling of performance with increase in number of processors is also generally infeasible, those best-case assumptions do not invalidate any of the conclusions on upper limits of achievable performance over the architectural configuration space. For example, for the modeled architectural parameters, the analysis indicates that achievable performance for large CG and FFT computations (that cannot fit within LLC) will necessarily be far below system peak, irrespective of the amount of effort put into the implementation, because fundamental inherent data movement limits of those computations cannot be overcome.

8.5 Algorithm-Architecture Codesign: Bounds on Energy Efficiency

An important metric is energy efficiency, defined as the ratio of number of executed operations to the energy expended in the execution. The upper bounds on $OI$ can also be used to bound the maximal achievable energy efficiency. The total energy of execution is modeled as the sum of energy for performing the operations, the energy for data movement from DRAM access, and an additional component for the static leakage energy in the cores and cache. Figure 8.5 shows the variation in the upper
bounds on energy efficiency for MM, FFT, CG and J2D as a function of: 1) number of cores used, 2) the voltage and clock frequency used, and 3) the capacity of LLC.

**Figure 8.5:** Upper bounds on energy efficiency.

The horizontal axis marks three different clock frequencies (2260 MHz, 1860 MHz, and 1460 MHz), representing voltage/frequency scaling, and for each of the frequencies, five choices of processor count (1, 2, 4, 8, and 16). Different curves correspond to different capacities of the LLC. The overall trends are as follows.

- For all four benchmarks, the groups of clustered lines move upwards as we go from left to right on the charts, representing a decrease in the voltage/frequency.
For a fixed number of cores and LLC capacity, lower frequencies lead to higher bounds on attainable energy efficiency. This is because there is a nonlinear decrease in the core’s energy per operation (energy is proportional to $V^2f$, and voltage and frequency are linearly related, so that energy is proportional to $f^3$) as voltage/frequency are decreased. There is also an increase in the total static leakage energy since the computation will take longer to complete, but there is an overall reduction in the lower bounds for energy, or an increase in the upper bounds on energy efficiency.

- Increasing the number of cores (with fixed frequency and LLC) is detrimental to energy efficiency, especially for bandwidth-bound computations. This is seen clearly for FFT and CG, with each of the lines curving downwards as the number of processors is increased. This effect is mainly due to the increased static energy for the active cores. While additional cores can divide the parallel work among themselves, the computation rate is limited by the rate at which data is delivered from memory to cache, so that there is no reduction in the lower bound for execution time. For MM and J2D, the curves are relatively flat since the computation is compute-bound. Using more cores enables the work to be done faster, and there is no increase in the total static leakage energy aggregated over all cores: using twice as many cores halves the lower bound on execution time and doubles the static leakage power of the cores.

- Increasing LLC capacity has two complementary effects: (i) potential for improving energy efficiency by increasing $OI$ due to the larger cache, but (ii)
decreased energy efficiency due to higher static leakage energy from the larger cache.

There is no useful benefit for CG since its $OI$ is independent of cache capacity. At larger cache sizes, there is a detrimental effect (not seen in the charts since the cache sizes used were quite small). For FFT, the benefits from improved $OI$ clearly outweigh the increased static leakage energy. For MM and J2D, although $OI$ increases with cache size, it is already so high at the smallest cache size that the incremental benefits in reducing data transfer energy from main memory are very small. Hence the curves representing different cache sizes for fixed frequency do not have much separation.

We next characterize the maximal possible energy efficiency for the four benchmarks, if we have the flexibility to choose (i) the number of cores to be turned on, (ii) the amount of cache area to be used, and (iii) voltage/frequency at which the cores are to be run. From Equations (8.1) and (8.2), we have the energy efficiency,

$$E_{\text{eff}} = \frac{W}{E} = \left( \epsilon_{\text{flop}}(f) + \frac{\epsilon_{\text{mem}}}{OI(S)} + \frac{P \cdot \pi_{\text{cpu}} + \pi_{\text{cache}}(S)}{\min(P \cdot F_{\text{flops}}(f), B_{\text{mem}}, OI(S))} \right)^{-1}. \quad (8.3)$$

Depending on the upper bound on the $OI$ of the algorithms, we analyze different cases below and determine what is the best configuration for the architecture to obtain maximum energy efficiency.

**Case I: The algorithm is completely bandwidth-bound.**

This corresponds to the case where the maximal $OI(S)$ of the algorithm for a given cache size $S$, is too low that it is bandwidth-bound even on a single core at its lowest frequency. We can see from the performance roofline viewpoint that this leads to the condition $B_{\text{mem}}, OI(S) < F_{\text{flops}}(f)$. With increasing frequency, $\epsilon_{\text{flop}}(f)$ increases and
thus the energy efficiency deteriorates. Hence, highest energy efficiency is achieved when $P = 1$ and $f$ is set at its minimum, and, Equation (8.3) reduces to

$$E_{\text{eff}} = \left(\epsilon_{\text{flop}}(f_{\text{min}}) + \frac{\epsilon_{\text{mem}}}{OI(S)} + \frac{\pi_{\text{cpu}} + \pi_{\text{cache}}(S)}{B_{\text{mem}}.OI(S)}\right)^{-1},$$

where, $f_{\text{min}}$ is the minimum allowable frequency for the architecture.

**Case II:** *The algorithm is compute-bound with $p$ or less cores and at all frequencies.*

From Equation (8.3), we can see that at any given frequency $f$, increasing $P$ improves the $E_{\text{eff}}$. Hence, $P = p$ is the best choice irrespective of the frequency. Also, with increasing $f$, $F_{\text{flops}}(f)$ increases linearly, while $\epsilon_{\text{flop}}(f)$ increases super-linearly. For a fixed value of $p$, the optimal energy efficiency is dependent on the machine parameters like $\pi_{\text{cache}}(S)$ and $\epsilon_{\text{flop}}(f)$. The maximal energy efficiency obtained,

$$E_{\text{eff}} = \max_{f \in [f_{\text{min}}, f_{\text{max}}]} \left(\epsilon_{\text{flop}}(f) + \frac{\epsilon_{\text{mem}}}{OI(S)} + \frac{p.\pi_{\text{cpu}} + \pi_{\text{cache}}(S)}{p.F_{\text{flops}}(f)}\right)^{-1},$$

where, $f_{\text{min}}$ and $f_{\text{max}}$ are the minimum and maximum allowable frequencies for the architecture, respectively.

**Case III:** *The algorithm is compute-bound with $p$ cores at lower frequencies and becomes bandwidth-bound with $p$ cores at higher frequencies.*

Let $f_{\text{cutoff}}$ be the frequency where the algorithm transitions from compute-bound to memory-bound region. For the region where $f \geq f_{\text{cutoff}}$, from case I, we know that once the algorithm becomes bandwidth-bound, increasing the frequency further has detrimental effect on the energy efficiency. Hence, the best energy efficiency is achieved when $f = f_{\text{cutoff}}$, where, $p.F_{\text{flops}}(f_{\text{cutoff}}) = B_{\text{mem}}.OI(S)$. When $f < f_{\text{cutoff}}$,
analysis in case II showed that in the compute-bound region when the number of cores, \( p \), is held constant, optimal frequency depends on the machine parameters. Also, we have, \( p.F_{\text{flops}}(f) < B_{\text{mem}}.OI(S) \). Hence,

\[
E_{\text{eff}} = \max_{f \in [f_{\text{min}}, f_{\text{cutoff}}]} \left( \epsilon_{\text{flop}}(f) + \frac{\epsilon_{\text{mem}}}{OI(S)} + \frac{p.\pi_{\text{cpu}} + \pi_{\text{cache}}(S)}{p.F_{\text{flops}}(f)} \right)^{-1}.
\]

**Case IV:** The algorithm is compute-bound at all frequencies with \( p \) cores, and becomes bandwidth bound with \( q = p + 1 \) or higher number of cores.

This case gives rise to two scenarios: (1) Performance at higher frequencies \((f_p \in [f_1, f_{\text{max}}])\) with \( p \) cores overlaps with the performance at lower frequencies \((f_q \in [f_{\text{min}}, f_2])\) with \( q \) cores, and hence, the algorithm becomes bandwidth-bound at frequencies \( f > f_2 \) and \( q \) cores. (2) There is a gap between maximum performance achievable with \( p \) cores and minimum performance achieved with \( q \) cores, i.e.,

\[(q.F_{\text{flops}}(f_{\text{min}}) - p.F_{\text{flops}}(f_{\text{max}})) > 0.\]

In both these scenarios, the maximum achievable energy efficiency depends on the machine parameters.

Similar to the performance roofline in Figure 8.4, an energy multi-roofline has been plotted for the four benchmarks in Figure 8.6 with a range of \( OI \) corresponding to the range of LLC sizes for our testbed architecture. Unlike the case for the performance multi-roofline, the actual set of energy roofline curves are not shown in the figure because there are too many of them and clutter the graph - in addition to different values of \( \alpha \), we also consider different core frequencies, resulting in a different energy roofline for each combination.
Figure 8.6: Upper bounds on energy efficiency

Figure 8.6 shows a similar trend for the energy efficiency as that of the performance in Section 8.4. MM and J2D are always compute-bound and hence fall under case II for all values of $S$. For MM and J2D, optimal energy efficiency was generally achieved at the lowest frequency value on the testbed. But, as the number of active cores approached one (as the size of LLC increases), maximal energy efficiency was achieved at higher frequencies. On the other hand, CG becomes bandwidth-bound as the number of active cores for the given LLC size exceeds 3. Hence, CG starts at case IV with $P = 3$, and as the LLC size increases (and the number of allowable cores goes below 3), it enters into case II. CG has the best upper bound on energy efficiency of 0.373 GFLOP/J for the cache size of 4 KB with 3 cores @ 1.26 GHz. The upper bound on energy efficiency for FFT initially increases with the increasing cache size and then starts decreasing for the similar reasons as that of the performance upper
bounds. FFT achieves its maximal energy efficiency for the cache size of 8 MB at $P = 10$ and $f = 1.26$ GHz.

8.6 Related Work

Williams et al. [55] developed the roofline model that attempts to analyze bottlenecks in performance of an architecture due to memory bandwidth limits. Choi et al. [13, 14] developed the energy version of the roofline model. Czechowski et al. [16] have developed balance principles for algorithm-architecture co-design. These models characterize algorithms using operational intensity of a particular implementation, which however is not a fixed quantity for an algorithm-architecture combination. Our work complements the above efforts. Demmel et al. [30] considered the energy efficiency at the algorithmic level and proved that a region of perfect strong scaling in energy exists for matrix multiplication and direct n-body problem. They also address the problem of finding the minimum energy required for a computation given a maximum allowed runtime and vice-versa. Bilardi et al. [10] developed an analysis of the tradeoffs between bandwidth, memory and processing for Quantum Chrome Dynamics (QCD) computation, and derived guidelines to design a machine for QCD computation.
CHAPTER 9

Conclusion and Future Work

In this dissertation, we developed new lower bounding techniques for characterizing data movement complexity of computations, presented heuristics for automatic analysis of computations for their data movement lower bounds, and demonstrated effectiveness of our techniques. We also presented an extension of the pebble game model to capture the data movement needs in inter-core/inter-node parallel environment. Several use cases and an application to algorithm-architecture codesign space exploration were also presented.

In Chapter 3, we developed a min-cut based lower bounding approach and provided tight lower bounds for different algorithms. Several of the past studies mainly focused on analytical reasoning and manual application of various techniques to derive lower bounds on data movement complexity. We showed that our technique lends itself to a more practical automatic heuristic. This also raises many interesting directions for future research. For instance, our automated min-cut based heuristic relies on good CDAG decomposition in order to obtain tight bounds. Our current implementation uses general-purpose graph partitioning algorithms to decompose CDAGs. However, a more sophisticated graph partitioning method that is guided by the lower
bounding heuristic to produce high quality decomposition would be of interest to pursue further.

The vast majority of existing application codes do not perform any redundant recomputation of any operations. But with data movement costs becoming increasingly dominant over operation execution costs, both in terms of energy and performance, there is significant interest in devising implementations of algorithms where redundant recomputation of values may be used to trade off additional inexpensive operations for a reduction in expensive data movements to/from off-chip memory. It is therefore of interest to develop automated techniques for data movement lower bounds under the original model of Hong & Kung that permits recomputation of CDAG vertices. Having lower bounds under both models can offer a mechanism to identify which algorithms have a potential for a trade-off between extra computations for reduced data movement and which do not. If the CDAG representing a computation has matching and tight data movement lower bounds under both the general model and the restricted model, the algorithm does not have potential for such a trade-off. On the other hand, if a lower bound under the restricted model is higher than a tight lower bound under the general model, the computation has potential for trading off extra computations for a reduction in volume of data movement. This raises an interesting question: Is it possible to develop necessary and/or sufficient conditions on properties of the computation, for example on the nature of the data dependencies, which will guarantee matching (or differing) lower bounds under the models allowing/prohibiting re-computation?
Chapter 6 built upon the work of [15] to develop technique to derive asymptotic parametric lower bounds for CDAGs of affine parts of computations. Many practical applications would benefit from having tight constants for the leading terms in the asymptotic parametric expressions of the order complexity of data movement lower bounds. Methods for finding tight constants for the parametric bounds are not addressed in this dissertation and would be an interesting topic for future research.

Deriving upper bounds that match the lower bounds are of significant importance to determine tightness of derived bounds and to develop actual implementations that achieve these bounds. Tightness of lower bounds in many cases are shown using an actual implementation with the data movement cost that matches the lower bound. There have been a few studies to develop techniques with matching lower and upper bounds [1, 9, 11, 12, 33]. An open question is whether any automatic tool can be designed to systematically explore the space of valid schedules to generate good upper bounds based on models and/or heuristics that minimize data movement cost.
BIBLIOGRAPHY


19 Elango, V., Rastello, F., Pouchet, L.-N., Ramanujam, J., and Sadayappan, P. On characterizing the data access complexity of programs. In


[33] Knight, N. S. Communication-optimal loop nests.

[34] Lemon Graph Library. https://lemon.cs.elte.hu.


