On Characterizing the Data Access Complexity of Programs

Venmugil Elango
The Ohio State University
elango.4@osu.edu

Fabrice Rastello
Inria
Fabrice.Rastello@inria.fr

Louis-Noël Pouchet
The Ohio State University
pouchet@cse.ohio-state.edu

J. Ramanujam
Louisiana State University
ram@cct.lsu.edu

P. Sadayappan
The Ohio State University
saday@cse.ohio-state.edu

Abstract

Technology trends will cause data movement to account for the majority of energy expenditure and execution time on emerging computers. Therefore, computational complexity will no longer be a sufficient metric for comparing algorithms, and a fundamental characterization of data access complexity will be increasingly important. The problem of developing lower bounds for data access complexity has been modeled using the formalism of Hong & Kung's red/blue pebble game for computational directed acyclic graphs (CDAGs). However, previously developed approaches to lower bounds analysis for the red/blue pebble game are very limited in effectiveness when applied to CDAGs of real programs, with computations comprised of multiple sub-computations with differing DAG structure. We address this problem by developing an approach for effectively composing lower bounds based on graph decomposition. We also develop a static analysis algorithm to derive the asymptotic data-access lower bounds of programs, as a function of the problem size and cache size.

Categories and Subject Descriptors F.2 [*Analysis of Algorithms and Problem Complexity*]: General; D.2.8 [*Software*]: Metrics—Complexity measures

General Terms Algorithms, Theory

Keywords Data access complexity; I/O lower bounds; Red-blue pebble game; Static analysis

1. Introduction

Advances in technology over the last few decades have yielded significantly different rates of improvement in the computational performance of processors relative to the speed of memory access. Because of the significant mismatch between computational latency and throughput when compared to main memory latency and bandwidth, the use of hierarchical memory systems and the exploitation of significant data reuse in the faster (i.e., higher) levels of the memory hierarchy is critical for high performance. With future systems, the cost of data movement through the memory hierarchy is expected to become even more dominant relative to the cost of performing arithmetic operations [6, 17, 32], both in terms of time and energy. It is therefore of critical importance to limit the volume of data movement to/from memory by enhancing data reuse in registers and higher levels of the cache. Thus the characterization of the inherent data access complexity of computations is extremely important.

```
for(i=1;i<N-1;i++)
  for(j=1;j<N-1;j++)
    A[i,j]=A[i-1,j]+A[i,j-1];
```
(a) Untiled code

```
for(it=1;it<N-1;it+=T)
  for(jt=1;jt<N-1;jt+=T)
    for(i=it;i<min(it+T,N-1);i++)
      for(j=jt;j<min(jt+T,N-1);j++)
        A[i,j]=A[i-1,j]+A[i,j-1];
```
(b) Equivalent tiled code

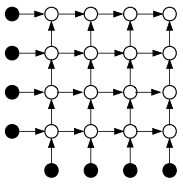

(c) CDAG

Figure 1: Single-sweep two-point Gauss-Seidel code

Let us consider the code shown in Fig. 1(a). Its computational complexity can be simply stated as $(N-2)^2$ arithmetic operations. Fig. 1(b) shows a functionally equivalent form of the same computation, after a tiling transformation. The tiled form too has exactly the same computational complexity of $(N-2)^2$ arithmetic operations. Next, let us consider the data access cost for execution of these two code forms on a processor with a single level of cache. If the problem size N is larger than cache capacity, the number of cache misses would be higher for the untiled version (Fig. 1(a)) than the tiled version (Fig. 1(b)). 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 valid orders of execution of its operations and also independent of machine parameters like cache

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2677010

size, the data access cost depends both on the cache capacity and the order of execution of the operations of the algorithm.

A fundamental question therefore is: *Given a computation 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?*

In order to model the range of valid scheduling orders for the operations of an algorithm, it is common to use the abstraction of the computational directed acyclic graph (CDAG), with a vertex for each instance of each computational operation, and edges from producer instances to consumer instances. Fig. 1(c) shows the CDAG for the codes in Fig. 1(a) and Fig. 1(b), for N=6; although the relative order of operations is different between the tiled and untiled versions, the set of computation instances and the producer-consumer relationships for the flow of data are exactly the same (special "input" vertices in the CDAG represent values of elements of A that are read before they are written in the nested loop).

While in general it is intractable to precisely answer the above fundamental question on the absolute minimum number of data transfers between main memory and caches/registers among all valid execution schedules of a CDAG, it is feasible to develop *lower bounds* on the optimal number of data transfers.

An approach to developing a lower bound on the minimal data movement for a computation in a two-level memory hierarchy was addressed in the seminal work of Hong & Kung by using the model of the red/blue pebble game on a computational directed acyclic graph (CDAG) [20]. While the approach has been used to develop I/O lower bounds for a small number of homogeneous computational kernels, as elaborated later, it poses challenges for effective analysis of full applications that are comprised of a number of parts with differing CDAG structure.

In this paper, we address the problem of analysis of affine loop programs to develop lower bounds on their data movement complexity. The work presented in this paper makes the following contributions:

- **Enabling composition in analysis of data access lower bounds**: It adapts the Hong & Kung pebble game model on CDAGs and the associated model of *S-partitioning* under a restriction that disallows recomputation, thereby enabling effective composition of I/O lower bounds for composite CDAGs from lower bounds for component CDAGs.
- **Static analysis of programs for lower bounds characterization**: It develops an approach for asymptotic parametric analysis of data-access lower bounds for arbitrary affine loop programs, as a function of cache size and problem size. This is done by analyzing linearly independent families of non-intersecting dependence chains.

2. Background

2.1 Computational Model

We are interested in modeling the inherent data access complexity of a computation, defined as the minimum number of data elements to be moved between local memory (with limited capacity but fast access by the processor) and main memory (much slower access but unlimited capacity) among all valid execution orders for the operations making up the computation. While the key developments in this paper can be naturally extended to address multi-level memory hierarchies and parallel execution, using an approach like the MMHG (Multiprocessor Memory Hierarchy Game) model of Savage & Zubair [29], we restrict the treatment in this paper to the case of only two levels of memory hierarchy and sequential execution.

The model of computation we use is a computational directed acyclic graph (CDAG), where computational operations are represented as graph vertices and the flow of values between operations is captured by graph edges. Fig. 2 shows an example of a CDAG corresponding to a simple loop program. Two important characteristics of this abstract form of representing a computation are that (1) there is no specification of a particular order of execution of the operations: although the program executes the operations in a specific sequential order, 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. (labels in Fig. 2 are only shown for aiding explanation; they are not part of the formal description of a CDAG).

```
for (i = 1; i < 4; ++i)
  S += A[i-1] + A[i];
```

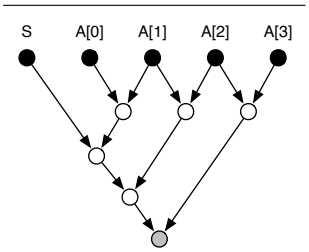


Figure 2: Example of a CDAG. Input vertices are represented in black, output vertices in grey.

We use the notation of Bilardi & Peserico [7] to formally describe the CDAG model used by Hong & Kung:

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

2.2 The Red-Blue Pebble Game

Hong & Kung used this computational model in their seminal work [20]. The inherent I/O complexity of a CDAG is the minimal number of I/O operations needed while optimally playing the *Red-Blue pebble game*. This game uses two kinds of pebbles: a fixed number of red pebbles that represent the small fast local memory (could represent cache, registers, etc.), and an arbitrarily large number of blue pebbles that represent the large slow main memory.

DEFINITION 2 (Red-Blue pebble game [20]). *Let $C = (I,V,E,O)$ be a CDAG such that any vertex with no incoming (resp. outgoing) edge is an element of I (resp. O). Given S red pebbles and an arbitrary number of blue pebbles, with an initial blue pebble on each input vertex, a complete calculation is any sequence of steps using the following rules that results in a final configuration with blue pebbles on all output vertices:*

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 immediate predecessors of a vertex $v \in V \setminus I$ have red pebbles, a red pebble may be placed on (or moved to)* [1] *v (execution or "firing" of operation),*

[1] The original red-blue pebble game in [20] 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 [27] have also considered a similar modification. But all our proofs hold for both the variants.

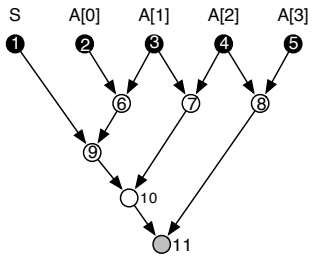


Figure 3: Example of schedule for a *complete calculation* on CDAG in Fig. 2. The vertex numbers represent the order of execution.

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

The number of I/O operations for any complete calculation is the total number of moves using rules R1 or R2, i.e., the total number of data movements between the fast and slow memories. The inherent I/O complexity of a CDAG is the smallest number of such I/O operations that can be achieved, among all complete calculations for that CDAG. An *optimal* calculation is a complete calculation achieving the minimum number of I/O operations.

Fig. 3 shows an example schedule for the CDAG in Fig. 2. Given S red pebbles and unlimited blue pebbles, goal of the game is to begin with blue pebbles on all input vertices, and finish with blue pebbles on all output vertices by following the rules in Definition 2 without using more than S red pebbles. Considering the case with three red pebbles ($S = 3$), one possible complete calculation for the CDAG in Fig. 3 is: $\{R1_2, R1_3, R3_6, R4_2, R1_1, R3_9, R4_1, R4_6, R1_4, R3_7, R4_3, R3_{10}, R4_9, R4_7, R1_5, R3_8, R4_4, R4_5, R3_{11}, R2_{11}\}$. The I/O cost of this complete calculation is 6 (which corresponds to the number of moves using rules $R1$ and $R2$). A different complete calculation for the same CDAG with I/O cost of 12 is given by: $\{R1_2, R1_3, R3_6, R4_2, R1_4, R2_6, R3_7, R4_3, R1_5, R2_7, R3_8, R2_8, R1_1, R1_6, R3_9, R4_1, R4_6, R1_7, R3_{10}, R4_7, R4_9, R1_8, R3_{11}, R2_{11}\}$. The I/O complexity of the CDAG is the minimum I/O cost of all such complete calculations.

2.3 Lower Bounds on I/O Complexity via *S*-Partitioning

While the red-blue pebble game provides an operational definition for the I/O complexity problem, it is generally not feasible to determine an optimal calculation on a CDAG. Hong & Kung developed a novel approach for deriving I/O lower bounds for CDAGs by relating the red-blue pebble game to a graph partitioning problem defined as follows.

DEFINITION 3 (Hong & Kung S-partitioning of a CDAG [20]). *Let $C = (I, V, E, O)$ be a CDAG. An S-partitioning of C is a collection of h subsets of V such that:*

P1 $\forall i \neq j,\ V_i \cap V_j = \emptyset$, *and* $\bigcup_{i=1}^{h} V_i = V$
P2 *there is no cyclic dependence between subsets*
P3 $\forall i,\ \exists D \in Dom(V_i)$ *such that* $|D| \leq S$
P4 $\forall i,\ |Min(V_i)| \leq S$

where a dominator set of V_i, $D \in Dom(V_i)$ is a set of vertices such that any path from I to a vertex in V_i contains some vertex in D; the minimum set of V_i, $Min(V_i)$ is the set of vertices in V_i that have all its successors outside of V_i; and for a set A, $|A|$ is the cardinality of the set A.

Hong & Kung showed a construction for a $2S$-partition of a CDAG, corresponding to any complete calculation on that CDAG using S red pebbles, with a tight relationship between the number of vertex sets h in the $2S$-partition and the number of I/O moves q in the complete calculation, as shown in Theorem 1. The tight association between any complete calculation and a corresponding $2S$-partition provides the key Lemma 1 that serves as the basis for Hong & Kung's approach for deriving lower bounds on the I/O complexity of CDAGs typically by reasoning on the maximal number of vertices that could belong to any vertex-set in a valid $2S$-partition.

THEOREM 1 (Pebble game, I/O and $2S$-partition [20]). *Any complete calculation of the red-blue pebble game on a CDAG using at most S red pebbles is associated with a $2S$-partition of the CDAG such that $S\,h \geq q \geq S\,(h-1)$, where q is the number of I/O moves in the complete calculation and h is the number of subsets in the $2S$-partition.*

LEMMA 1 (Lower bound on I/O [20]). *Let H be the minimal number of vertex sets for any valid $2S$-partition of a given CDAG (such that any vertex with no incoming – resp. outgoing – edge is an element of I – resp. O). Then the minimal number Q of I/O operations for any complete calculation on the CDAG is bounded by: $Q \geq S \times (H-1)$*

This key lemma has been useful in proving I/O lower bounds for several CDAGs [20] by reasoning about the maximal number of vertices that could belong to any vertex-set in a valid $2S$-partition.

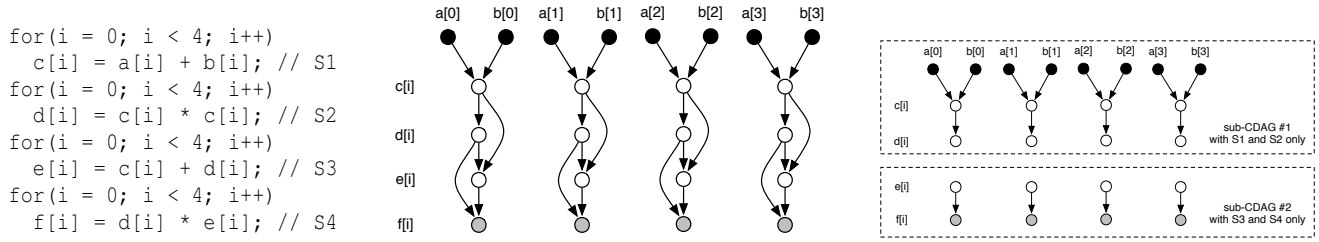
3. Challenges in Composing I/O Lower Bounds from Partitioned CDAGs

Application codes are typically constructed from a number of sub-computations using the fundamental composition mechanisms of sequencing, iteration and recursion. As explained in Sec. 1, in contrast to analysis of computational complexity of such composite application codes, I/O complexity analysis poses challenges. With computational complexity, the operation counts of sub-computations can simply be added. However, using the red/blue pebble game model of Hong & Kung, as elaborated below, it is problematic to analyze the I/O complexity of sub-computations and simply combine them by addition. In the next section, we develop an approach to overcome the problem.

3.1 The Decomposition Problem

The Hong & Kung red/blue pebble game model places blue pebbles on all CDAG vertices without predecessors, since such vertices are considered to hold inputs to the computation, and therefore assumed to start off in slow memory. Similarly, all vertices without successors are considered to be outputs of the computation, and must have blue pebbles at the end of a complete calculation. If the vertices of a CDAG corresponding to a composite application are disjointly partitioned into sub-DAGs, the analysis of each sub-DAG will require the initial placement of blue pebbles on all vertices without predecessors in the sub-DAG, and final placement of blue pebbles on all vertices without successors in the sub-DAG. So an optimal calculation for each sub-DAG will require at least one load (R1) operation for each input and a store (R2) operation for each output. But in a complete calculation on the full composite CDAG, clearly it may be possible to pass values in a red pebble between vertices in different sub-DAGs, so that the I/O complexity could be less than the sum of the I/O costs for optimal calculations on each sub-DAG. This is illustrated by the following example.

Fig. 4(b) shows the CDAG for the computation in Fig. 4(a). Fig. 4(c) shows the CDAG partitioned into two sub-DAGs, where the first sub-DAG contains vertices of $S1$ and $S2$ (and the input vertices corresponding to `a[i]` and `b[i]`), and the second sub-DAG contains vertices of $S3$ and $S4$. Considering the full CDAG, with just two red pebbles, it can be computed at an I/O cost of 12, incurring I/O just for the initial loads of inputs `a[i]` and `b[i]`, and the final stores for outputs `f[i]`. In contrast, with the partitioned sub-DAGs, the first sub-DAG will incur additional output stores



(a) Original code (b) Full CDAG (c) CDAG partitioning

Figure 4: Example illustrating limitation of Hong & Kung model regarding composition of lower bounds from sub-components of CDAG

for the successor-free vertices $S2[i]$, and the second sub-DAG will incur input loads for predecessor-free vertices $S3[i]$. Thus the sum of optimal red/blue pebble game I/O costs for the two sub-DAGs amounts to 20 moves, i.e., it exceeds the optimal I/O cost for the full CDAG.

The above example illustrates a fundamental problem with the Hong & Kung red/blue pebble game model: a simple combining of I/O lower bounds for sub-DAGs of a CDAG cannot be used to generate an I/O lower bound for the composite CDAG. But the ability to perform complexity analysis by combining analyses of component sub-computations is important for the analysis of real applications. Such decomposition of data-access complexity analysis can be enabled by making a change to the Hong & Kung pebble game model, as discussed next.

3.2 Flexible Input/Output Vertex Labeling to Enable Composition of Lower Bounds

With the Hong & Kung model, all vertices without predecessors must be input vertices, and all vertices without successors must be output vertices. By relaxing this constraint, we show that composition of lower bounds from sub-CDAGs is valid. With such a modification, vertices without predecessors will not be required to be input vertices, and such predecessor-free non-input vertices do not have an initial blue pebble placed on them. However, such vertices are allowed to fire using rule R3 at any time, since they do not have any predecessor nodes without red pebbles. Vertices without successors are similarly not required to be output vertices, and those not designated as outputs do not need a blue pebble on them at the end of the game. However, all compute vertices (i.e., vertices in $V \setminus I$) in CDAG are required to have fired for any complete calculation.

Using the modified model of the red/blue pebble game with flexible input/output vertex labeling, it is feasible to compose I/O lower bounds by adding lower bounds for disjointly partitioned sub-CDAGs of a CDAG. The following theorem formalizes it.

THEOREM 2 (Decomposition). *Let $C = (I,V,E,O)$ be a CDAG. Let $\{V_1,V_2,\ldots,V_p\}$ be an arbitrary (not necessarily acyclic) disjoint partitioning of V ($\bigcap_{i=1}^{p} V_i = \emptyset$ and $\bigcup_{i=1}^{p} V_i = V$) and $C_1,C_2,\ldots,C_p$ be the induced partitioning of C ($I_i = I \cap V_i$, $E_i = E \cap V_i \times V_i$, $O_i = O \cap V_i$). If Q is the I/O complexity for C and Q_i is the I/O complexity for C_i, then $\sum_{i=1}^{p} Q_i \leq Q$. In particular, if L_i is the I/O lower bound for C_i, then $\sum_{i=1}^{p} L_i$ is an I/O lower bound for C.*

Proof. Consider an optimal calculation $\mathcal{P}$ for C, with cost Q. We define the cost of $\mathcal{P}$ restricted to V_i, denoted as $Q_{|V_i}$, as the number of $R1$ or $R2$ transitions in $\mathcal{P}$ that involve a vertex of V_i. Clearly $Q = \sum_{i=1}^{p} Q_{|V_i}$. We will show that we can build from $\mathcal{P}$, a valid complete calculation $\mathcal{P}_{|V_i}$ for C_i, of cost $Q_{|V_i}$. This will prove that $Q_i \leq Q_{|V_i}$, and thus $\sum_{i=1}^{p} Q_i \leq \sum_{i=1}^{p} Q_{|V_i} = Q$. $\mathcal{P}_{|V_i}$ is built from $\mathcal{P}$ as follows: (1) for any transition in $\mathcal{P}$ that involves a vertex $v \in V_i$, apply this transition in $\mathcal{P}_{|V_i}$; (2) delete all other transitions in $\mathcal{P}$. Conditions for transitions $R1$, $R2$, and $R4$ are trivially satisfied. Whenever a transition $R3$ on a vertex v is performed in $\mathcal{P}$, all the predecessors of v must have a red pebble on them. Since all transitions of $\mathcal{P}$ on the vertices of V_i are maintained in $\mathcal{P}_{|V_i}$, when v is executed in $\mathcal{P}_{|V_i}$, all its predecessor vertices must have red pebbles, enabling transition $R3$. □

With this modified model of the red/blue pebble game that permits predecessor-free vertices to be non-input vertices, complex CDAGs can be decomposed and lower bounds for the composite CDAG can be obtained by composition of the bounds from the sub-CDAGs. However, sub-CDAGs that have no "true" input and output vertices in them will have trivial I/O lower bounds of zero – the entire set of vertices in the sub-CDAG can fit in a single vertex set for a valid 2S-partition, for any value of S, since conditions **P1-P4** are trivially satisfied.

In the next section, we present a solution to the problem. The main idea is to impose restrictions on the red/blue pebble game to disallow re-pebbling or multiple firings of any vertex using rule R3. We show that by imposing such a restriction, we can develop an input/output tagging strategy for sub-CDAGs that enables stronger lower bounds to be generated by CDAG decomposition.

4. S-Partitioning when Re-Pebbling is Prohibited

With the pebble game model of Hong & Kung, the compute rule R3 could be applied multiple times in a complete calculation. This is useful in modeling algorithms that perform re-computation of multiply used values rather than incur the overhead of storing and loading it. However, the majority of practically used algorithms do not perform any redundant recomputation. Hence several efforts [1, 3, 7, 10, 12, 21, 24–28, 30, 31] have modeled I/O complexity under a more restrictive model that disallows recomputation, primarily because it eases or enables analysis with some lower bounding techniques. In this section, we consider the issue of composing bounds via CDAG decomposition under a model that disallows recomputation, i.e., prohibits re-pebbling. We develop a modified definition of S-partition that is adapted to enable I/O lower bounds to be developed for the restricted red/blue pebble game. This provides two significant benefits:

1. It enables non-trivial I/O lower bound contributions to be accumulated from sub-CDAGs of a CDAG, even when the sub-CDAGs do not have any true inputs. This is achieved via input/output tagging/untagging strategies we develop in this section.
2. It enables static analysis of programs to develop parametric expressions for asymptotic lower bounds as a function of cache and problem size parameters. This is described in the following sections.

A pebble game model that does not allow recomputation can be formalized by changing rule R3 of the red/blue pebble game

to R3-NR (NR denotes No-Recomputation or No-Repebbling) and the definition of a complete calculation as follows:

DEFINITION 4 (Recompute-restricted Red-Blue pebble game). *Let $C = (I,V,E,O)$ be a CDAG. Given S red pebbles and arbitrary number of blue pebbles, with an initial blue pebble on each input vertex, a complete calculation is any sequence of steps using the following rules that causes each vertex in $V \setminus I$ to be fired once using Rule R3-NR, and results in a final configuration with blue pebbles on all output vertices:*

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-NR (Compute) *If all immediate predecessors of a vertex $v \in V \setminus I$ have red pebbles on them, and a red pebble has not previously been placed on v, a red pebble may be placed on v.*

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

We next present an adaptation of Hong and Kung's S-partition that will enable us to develop larger lower bounds for the restricted pebble game model that prohibits repebbling.

DEFINITION 5 (S^{NR}-partitioning of CDAG). *Given a CDAG C, an S^{NR}-partitioning of C is a collection of h subsets of $V \setminus I$ such that:*

P1 $\forall i \neq j,\ V_i \cap V_j = \emptyset$, *and* $\bigcup_{i=1}^{h} V_i = V \setminus I$

P2 *there is no cyclic dependence between subsets*

P3 $\forall i,\ |In(V_i)| \leq S$

P4 $\forall i,\ |Out(V_i)| \leq S$

where the input set of V_i, $In(V_i)$ is the set of vertices of $V \setminus V_i$ that have at least one successor in V_i; the output set of V_i, $Out(V_i)$ is the set of vertices of V_i that are also part of the output set O or that have at least one successor outside of V_i.

THEOREM 3 (Restricted pebble game, I/O and $2S^{NR}$-partition). *Any complete calculation of the red-blue pebble game, without repebbling, on a CDAG using at most S red pebbles is associated with a $2S^{NR}$-partition of the CDAG such that $S \times h \geq q \geq S \times (h-1)$, where q is the number of I/O moves in the game and h is the number of subsets in the $2S^{NR}$-partition.*

Proof. Consider a complete calculation $\mathcal{P}$ that corresponds to some scheduling (i.e., execution) of the vertices of the graph $G = (V,E)$ that follows the rules R1–R4 of the restricted pebble game. We view this calculation as a string that has recorded all the transitions (applications of pebble game rules). Suppose that $\mathcal{P}$ contains exactly q transitions of type $R1$ or $R2$. Let $(\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_h)$ correspond to a partitioning of the transitions of $\mathcal{P}$ into $h = \lceil q/S \rceil$ consecutive sub-sequences such that each $\mathcal{P}_i \in (\mathcal{P}_1, \ldots, \mathcal{P}_{h-1})$ contains exactly S transitions of type $R1$ or $R2$.

The CDAG contains no node isolated from the output nodes, and any vertex of $V \setminus I$ is computed exactly once in $\mathcal{P}$. Let V_i be the set of vertices computed (transition R3-NR) in the sub-calculation $\mathcal{P}_i$. Property $P1$ is trivially fulfilled.

As transition R3-NR on a vertex v is possible only if its predecessor vertices have red pebbles on them, those predecessors are necessarily executed in some $\mathcal{P}_j$, $j \leq i$ and are thus part of a V_j, $j \leq i$. This proves property $P2$.

To prove $P3$, for a given V_i we consider two sets: V_R is the set of vertices that had a red pebble on them just before the execution of P_i; V_{BR} is the set of vertices on which a red pebble is placed according to rule $R1$ (input) during $\mathcal{P}_i$. We have, $\mathsf{In}(V_i) \subseteq V_R \cup V_{BR}$. Thus $|\mathsf{In}(V_i)| \leq |V_R| + |V_{BR}|$. As there only S red pebbles, $|V_R| \leq S$. Also by construction of $\mathcal{P}_i$, $|V_{RB}| \leq S$. This proves that $|\mathsf{In}(V_i)| \leq 2S$ (property $P3$).

Property $P4$ is proved in a similar way: V'_R is the set of vertices that have a red pebble on them just after the execution of $\mathcal{P}_i$; V'_{RB} is the set of vertices of V_i on which a blue pebble is placed during $\mathcal{P}_i$ according to rule $R2$. We have that $\mathsf{Out}(V_i) \subseteq V'_R \cup V'_{RB}$. Thus $|\mathsf{Out}(V_i)| \leq |V'_R| + |V'_{RB}|$. As there are only S red pebbles, $|V'_R| \leq S$. Also by construction of $\mathcal{P}_i$, $|V'_{RB}| \leq S$. This proves that $|\mathsf{Out}(V_i)| \leq 2S$ (property $P4$). □

LEMMA 2 (I/O lower bound for restricted pebble game). *Let H^{NR} be the minimal number of vertex sets for any valid $2S^{NR}$-partition of a given CDAG. Then the minimal number Q of I/O operations for any complete calculation on the CDAG, without any repebbling, is bounded by: $Q \geq S \times (H^{NR} - 1)$*

The above theorem and lemma establish the relationship between complete calculations of the restricted pebble game and 2S-NR partitions. The critical difference between the standard S-partition of Hong & Kung and the S-NR partition is the validity condition pertaining to incoming edges into a vertex set in the partition: for the former the size of dominator sets is constrained to be no more than S, while for the latter the number of external vertices with edges into the vertex set is constrained by S. When a CDAG is decomposed into sub-CDAGs, very often some of the sub-CDAGs get isolated from the CDAG's input and output vertices. This will lead to trivial (i.e., zero) lower bounds for such component sub-CDAGs. For the restricted pebble game, below we develop an approach to obtain tighter lower bounds for component sub-CDAGs that have become isolated from inputs and outputs of the full CDAG. The key idea is to allow any vertex without predecessors (resp. successors) to simulate an input (resp. output) vertex by specially tagging it so, and then adjusting the obtained lower bound to account for a one-time access cost for loading (resp. storing) such a tagged input (resp. output). The vertices of a CDAG remain unchanged, but the labeling (tag) of some vertices as inputs/outputs in the CDAG is changed.

THEOREM 4 (Input/Output (Un)Tagging – Restricted pebble game). *Let C and C' be two CDAGs of the same DAG $G = (V,E)$: $C = (I,V,E,O)$, $C' = (I \cup dI, V, E, O \cup dO)$, where, $dI \subseteq V$ and $dO \subseteq V$. If Q is the I/O complexity for C and Q' is the I/O complexity for C' then, Q can be bounded by Q' as follows (tagging):*

$$Q' - |dI| - |dO| \leq Q \tag{1}$$

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

$$Q \leq Q' \tag{2}$$

Proof. Consider an optimal calculation $\mathcal{P}$ for C, of cost Q. We will build a valid complete calculation $\mathcal{P}'$ for C', of cost no more than $Q + |dI| + |dO|$. This will prove that $Q' \leq Q + |dI| + |dO|$. We build $\mathcal{P}'$ from $\mathcal{P}$ as follows: (1) for any input vertex $v \in dI$, the (only) transition $R3$ involving v in $\mathcal{P}$ is replaced in $\mathcal{P}'$ by a transition $R1$; (2) for any output vertex $v \in dO$, the (only) transition $R3$ involving v in $\mathcal{P}$ is complemented by an $R2$ transition; (3) any other transition in $\mathcal{P}$ is reported as is in $\mathcal{P}'$.

Consider now an optimal calculation $\mathcal{P}'$ for C', of cost Q'. We will build a valid complete calculation $\mathcal{P}$ for C, of cost no more than Q'. This will prove that $Q \leq Q'$. We build $\mathcal{P}$ from $\mathcal{P}'$ as follows: (1) for any input vertex $v \in dI$, the first transition $R1$ involving v in $\mathcal{P}'$ is replaced in $\mathcal{P}$ by a transition $R3$ followed by a transition $R2$; (2) any other transition in $\mathcal{P}'$ is reported as is in $\mathcal{P}$. □

We note that such a construction is only possible for the restricted pebble game where repebbling is disallowed. It enables tighter lower bounds to be developed via CDAG decomposition. In the next section, we use S-NR partitioning and the untagging theorem in developing a static analysis approach to characterizing data-access lower bounds of loop programs.

5. Parametric Lower Bounds via Static Analysis of Programs

In this section, we develop a static analysis approach to derive asymptotic parametric I/O 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 [18]. Second, there exist readily available tools to perform complex geometric operations on such sets and relations. We use the Integer Set Library (ISL) [36] for our analysis.

In Subsection 5.1, we provide a description of the program representation for affine programs. In Subsection 5.2, we detail the geometric reasoning that is the basis for the developed I/O lower bounds approach. Subsection 5.3 describes the I/O lower bound computation using examples.

5.1 Background and Program Representation

In the following, we use ISL terminology [36] and syntax to describe sets and relations. We now recall some key concepts to represent program features.

Iteration domain A computation vertex in a CDAG represents a dynamic instance of some operation in the input program. For example, given a statement $S1$: `A[i] += B[i+1]` surrounded by one loop `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) $\mathbb{Z}$-polyhedra, i.e., a set of integer points bounded by affine inequalities intersected with an affine integer lattice [19]. Using ISL notation, the iteration domain of statement $S1$, D_{S1}, is denoted: `[n]->{S1[i]:0<=i<n}`. The left-hand side of `->`, `[n]` in the example, is the list of all parameters needed to define the set. `S1[i]` models a set with one dimension (`[i]`) named i, and the set space is named `S1`. 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) [4], and parametric (integer) linear programming [16] 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 I/O, an under-approximation of the statement domain and/or the set of dependences is acceptable, since an I/O lower bound for the approximated system is a valid lower bound for the actual system. For instance, if the iteration domain 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 $\underline{D_S} \subseteq D_S$. Such a polyhedron can be obtained, for instance, by first computing the convex hull $\overline{D_S} \supseteq D_S$ and then shifting its faces until they are strictly included in 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 [22].

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 $S1$, and the relation $R1$ describing this access is: `[n]->{S1[i] -> B[i+1] : 0<=i<n}`. This relation models a single edge between each element of set $S1$ and an element of set B, described by the relationship $i \rightarrow i+1$. Several operations on relations, such as $\mathsf{domain}(R)$, which computes the domain (e.g., input set) of the relation ($\mathsf{domain}(R1) =$ `[n] -> {[i] : 0<=i<n}`), $\mathsf{image}(R)$ computing the image (e.g., range, or output set) of R ($\mathsf{image}(R1) =$ `[n] -> {[i] : 1<=i<n+1}`), the composition of two relations $R1 \circ R2$, 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 $S1$ and $S2$ with a producer-consumer relationship, the edges connecting each dynamic instance of $S1$ and $S2$ in a CDAG can be expressed using relations. For example, `[n]->{S1[i,j] -> S2[i,j-1,k] : ...}` models a relation between a 2D statement and a 3D statement. Each point in $S1$ is connected to several points in $S2$ 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 I/O requirement), and therefore we only consider must-dependences in our analysis framework.

5.2 Geometric Reasoning for I/O Lower Bounds by 2S-partitioning

Given a CDAG, Lemma 2 establishes a relation between a lower bound on its data movement complexity for execution with S fast storage elements and the minimal possible number of vertex sets among all valid $2S^{NR}$-partitions of the CDAG. The minimum possible number of vertex sets in a $2S^{NR}$-partition is inversely related to the largest possible size of any vertex set for a valid $2S^{NR}$-partition. A geometric reasoning based on the Loomis-Whitney inequality [23] and its generalization [5, 34] has been used to establish I/O lower bounds for a number of linear algebra algorithms [1, 2, 11, 21]. A novel approach to determining I/O lower bounds for affine computations in perfectly nested loops has been recently developed [11] using similar geometric reasoning. The approach developed in this paper is inspired by that work and also uses a similar geometric reasoning, but improves on the prior work in two significant ways:

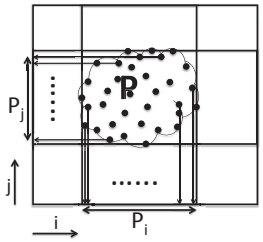
1. *Generality*: It can be applied to a broader class of computations, handling multiple statements and imperfectly nested loops.
2. *Tighter Bounds*: For computations with loop-carried dependences that are not oriented perfectly along one of the iteration space dimensions, it provides tighter I/O lower bounds, as illustrated by the Jacobi example in the next section.

Before presenting the details of the static analysis for lower bounds characterization of affine computations, we use a simple example to illustrate the geometric approach based on the Loomis-Whitney inequality and its generalizations that have been used to develop I/O lower bounds for matrix-multiplication and other linear algebra

computations. Consider the code exemplifying an N-body force calculation in Fig. 5(a). We have a 2D iteration space with $\Theta(N^2)$ points. The net force on each of N particles from the other particles is computed using the function `f()`, which uses the mass and position of a pair of particles to compute the force between them. The total number of input data elements for the computation is $\Theta(N)$. If $S < N$, it will be necessary to bring in at least some of the input data elements more than once from slow to fast memory. A geometric reasoning for a lower bound on the amount of I/O to/from fast memory proceeds as follows. Consider an arbitrary vertex set from any valid $2S^{NR}$-partition. Let the set of points P in the iteration space, illustrated by a cloud in Fig. 5(b), denote the vertex set. The projections of each of the points onto the two iteration space axes are shown. Let $|\mathsf{P}_i|$ and $|\mathsf{P}_j|$ respectively denote the number of distinct points on the i and j axes. $|\mathsf{P}_i|$ represents the number of distinct elements of input arrays `pos` and `mass` that are accessed in the computation, for references `pos(i)` and `mass(i)`. Similarly, $|\mathsf{P}_j|$ corresponds to the number of distinct elements accessed via the references `pos(j)` and `mass(j)`. For any vertex set from a valid $2S^{NR}$-partition, the size of the input set cannot exceed $2S$. Hence $2 \times |\mathsf{P}_i| \leq 2S$ and $2 \times |\mathsf{P}_j| \leq 2S$. For this 2D example, the Loomis-Whitney inequality asserts that the number of points in P cannot exceed $|\mathsf{P}_i| \times |\mathsf{P}_j|$. Combining the two inequalities, we can conclude that S^2 is an upper bound on the size of the vertex set. Thus, the minimum number of vertex sets in a valid 2S-partition, $H = \Omega(N^2/S^2)$. By Lemma 2, a lower bound on I/O is $(H-1) \times S$, i.e., $\Omega(N^2/S)$.

```
for(i=0;i<N;i++)
 for(j=0;j<N;j++)
  if (i <> j) force(i)
  += f(mass(i),mass(j),pos(i),pos(j));
```

(a) Code for N-body force calculation



(b) Geometric Projection

Figure 5: Illustration of Geometric Reasoning for I/O Lower Bounds

More generally, for a d-dimensional iteration space, given some bounds on the number of elements on some projections of P, a bound on $|\mathsf{P}|$ can be derived using a powerful approach developed by Christ et al. [11]. Christ et al. [11, Theorem 3.2] extended the discrete case of the Brascamp-Lieb inequality [5, Theorem 2.4] to obtain these bounds. Since our goal here is to develop asymptotic parametric bounds, the extension of the continuous Brascamp-Lieb inequality, stated below (in the restricted case of orthogonal projections and using the Lebesgue measure for volumes), is sufficient for our analysis. We use the notation $H \leq \mathbb{R}^d$ to denote that H is a linear subspace of $\mathbb{R}^d$.

THEOREM 5. *Let $\phi_j : \mathbb{R}^d \rightarrow \mathbb{R}^{d_j}$ 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 $(s_1,\ldots,s_m) \in [0,1]^m$:

$$\forall H \leq \mathbb{R}^d,\ \dim(H) \leq \sum_{j=1}^{m} s_j \dim(\phi_j(H)) \tag{3}$$

$$\iff \qquad \forall E \subseteq \mathbb{R}^d,\ |E| \leq \prod_{j=1}^{m} |\phi_j(E)|^{s_j} \tag{4}$$

Since the linear transformations ϕ_j are orthogonal projections, the following Theorem enables us to limit the number of inequalities of Eq. (3) required for Theorem 5 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):

THEOREM 6. *Let $\phi_j : \mathbb{R}^d \rightarrow \mathbb{R}^{d_j}$ 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 $(s_1,\ldots,s_m) \in [0,1]^m$:

$$\forall H \leq \mathbb{R}^d,\ \dim(H) \leq \sum_{j=1}^{m} s_j \dim(\phi_j(H)) \tag{5}$$

$$\iff \qquad \forall H_i = \langle e_i \rangle,\ 1 = \dim(H_i) \leq \sum_{j=1}^{m} s_i \delta_{i,j} \tag{6}$$

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

The proof of Theorem 6 directly corresponds to the proof of [11, Theorem 6.6] and is omitted (see also [5, Prop. 7.1]). It shows that if $s = (s_1,\ldots,s_m) \in [0,1]^m$ are such that $\forall H_i,\ 1 \leq \sum_{j=1}^{m} s_i \delta_{i,j}$, then the volume of any measurable set $E \subseteq \mathbb{R}^d$ can be bounded by $U_s = \prod_{j=1}^{m} |\phi_j(E)|^{s_j}$. In order to obtain as tight an asymptotic bound as possible, we seek s such that U_s is as small as possible. Since we have $|\phi_j(H)| \leq S$, this corresponds to finding s_j such that $\prod_{j=1}^{m} S^{s_j}$ is minimized, or equivalently, $S^{\sum_{j=1}^{m} s_j}$ is minimized. In other words, $\sum_{j=1}^{m} s_j$ has to be minimized. For this purpose, if $\forall i,\ \exists j,$ s.t., $\delta_{i,j} = 1$, we solve:

$$\text{Minimize } \sum_{j=1}^{m} s_j,\ \text{s.t., } \forall i,\ 1 \leq \sum_{j=1}^{m} s_j \delta_{i,j} \tag{7}$$

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

$$\text{Maximize } \sum_{i=1}^{d} x_i,\ \text{s.t., } \forall j,\ \sum_{i=1}^{d} x_i \delta_{i,j} \leq 1 \tag{8}$$

We use an illustrative example:

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

Consider the following three projections (we explain how the projection directions are obtained later in this section)

$$\phi_1 : (i,j,k) \rightarrow (i,j);\ \phi_2 : (i,j,k) \rightarrow (i,k);\ \phi_3 : (i,j,k) \rightarrow (k)$$

Let H_1, H_2 and H_3 denote the three subspaces spanned by the canonical bases of $\mathbb{R}^3$. Consider, for example, the linear map ϕ_1. We have $\delta_{1,1} = \dim(\phi_1(h_1)) = 1$ for any $h_1 \in H_1$, $\delta_{2,1} = \dim(\phi_1(h_2)) = 1$ for any $h_2 \in H_2$, and $\delta_{3,1} = \dim(\phi_1(h_3)) = 0$ for any $h_3 \in H_3$. Thus, we obtain the constraint $x_1.1 + x_2.1 + x_3.0 \leq 1$, or $x_1 + x_2 \leq 1$. Similarly, we obtain the remaining two constraints for the projections ϕ_2 and ϕ_3.

This results in the following linear programming problem:

$$\text{Maximize } x_1 + x_2 + x_3 \tag{9}$$

s.t. $x_1 + x_2 \leq 1;\ x_1 + x_3 \leq 1;\ x_3 \leq 1$

Solving Eq. (9) provides the solution $(x_1, x_2, x_3) = (0, 1, 1)$, i.e., $x_1 + x_2 + x_3 = 2$. The solution corresponds to considering a cube of asymptotic dimensions $1 \times S \times S$ and volume $O(S^{\Sigma_{j=1}^{3} x_j}) = O(S^2)$ as the largest vertex-set. This provides an I/O lower bound of $\Omega(N^3/S)$, when the problem size N is sufficiently large.

5.3 Automated I/O Lower Bound Computation

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

Illustrative example 1: Consider the following example of Jacobi 1D stencil computation.

```
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++)
S2:    B[i] = A[i-1] + A[i] + A[i+1];

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

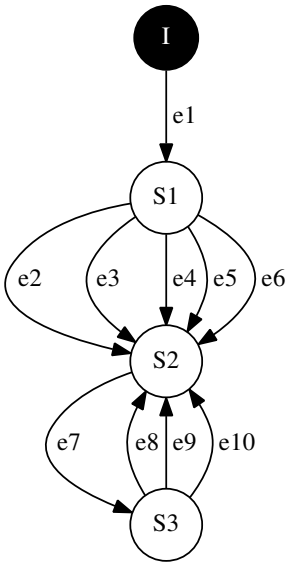


Figure 6: Data-flow graph for Jacobi 1D

Fig. 6 shows the static data-flow graph $G_F = (V_F, E_F)$ for Jacobi 1D. G_F contains a vertex for each statement in the code. The input array `I` is also explicitly represented in G_F by node I (shaded in black in Fig. 6). Each vertex has an associated domain as shown below:

- D_I = `[N]->{I[i]:0<=i<N}`
- D_{S1} = `[N]->{S1[i]:0<=i<N}`
- D_{S2} = `[T,N]->{S2[t,i]:1<=t<T and 1<=i<N-1}`
- D_{S3} = `[T,N]->{S3[t,i]:1<=t<T and 1<=i<N-1}`

The edges represent the true (read-after-write) data dependences between the statements. Each edge has an associated affine dependence relation as shown below:

- Edge $e1$: This edge corresponds to the dependence due to copying the inputs `I` to array `A` at statement $S1$ and has the following relation.
  `[N]->{I[i]->S1[i]:0<=i<N}`
- Edges $e2$, $e3$ and $e4$: The use of array elements `A[i-1]`, `A[i]` and `A[i+1]` at statement $S2$ are captured by edges $e2$, $e3$ and $e4$, respectively.
  `[T,N]->{S1[i]->S2[1,i+1]:1<=i<N-2}`
  `[T,N]->{S1[i]->S2[1,i]:1<=i<N-1}`
  `[T,N]->{S1[i]->S2[1,i-1]:2<=i<N-1}`
- Edges $e5$ and $e6$: Multiple uses of the boundary elements `I[0]` and `I[N-1]` by `A[t][1]` and `A[t][N-2]`, respectively, for `1<=t<T` are represented by the following relations.
  `[T,N]->{S1[0]->S2[t,1]:1<=t<T}`
  `[T,N]->{S1[N-1]->S2[t,N-2]:1<=t<T}`
- Edge $e7$: The use of array `B` in statement `S3` corresponds to edge $e7$ with the following relation.
  `[T,N]->{S2[t,i]->S3[t,i]:1<=t<T and 1<=i<N-1}`
- Edges $e8$, $e9$ and $e10$: The uses of array `A` in statement $S2$ from $S3$ are represented by these edges with the following relations.
  `[T,N]->{S3[t,i]->S2[t+1,i+1]:1<=t<T-1 and 1<=i<N-2}`
  `[T,N]->{S3[t,i]->S2[t+1,i]:1<=t<T-1 and 1<=i<N-1}`
  `[T,N]->{S3[t,i]->S2[t+1,i-1]:1<=t<T-1 and 2<=i<N-1}`

Given a path $p = (e_1, \ldots, e_l)$ with associated edge relations $(R_1, \ldots, R_l)$, the relation associated with p can be computed by composing the relations of its edges, i.e., $\mathsf{relation}(p) = R_l \circ \cdots \circ R_1$. For instance, the relation for the path $(e7, e8)$ in the example, obtained through the composition $R_{e8} \circ R_{e7}$, is given by R_p = `[T,N] -> {S2[t,i] -> 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., $\mathsf{domain}(R_j \circ R_i) = R_i^{-1}(\mathsf{image}(R_i) \cap \mathsf{domain}(R_j))$ and $\mathsf{image}(R_j \circ R_i) = R_j(\mathsf{image}(R_i) \cap \mathsf{domain}(R_j))$. Hence, $\mathsf{domain}(R_p)$ = `[T,N] -> {S2[t,i] : 1<=t<T-1 and 1<=i<N-2}` and $\mathsf{image}(R_p)$ = `[T,N] -> {S2[t,i] : 2<=t<T and 2<=i<N-1}`.

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

DEFINITION 6 (Injective edge and circuit). *An* injective edge *a is an edge of a data-flow graph whose associated relation R_a is both affine and injective, i.e., $R_a = \mathbf{A}.\vec{x} + \vec{b}$, where $\mathbf{A}$ is an invertible matrix. An* injective circuit *is a circuit E of a data-flow graph such that every edge $e \in E$ is an injective edge.*

DEFINITION 7 (Broadcast edge and path). *A* broadcast edge *b is an edge of a data-flow graph whose associated relation R_b is affine and* $\dim(\mathsf{domain}(R_b)) < \dim(\mathsf{image}(R_b))$. *A* broadcast path *is a path $(e_1, \ldots, e_n)$ of a data-flow graph such that e_1 is a broadcast edge and $\forall_{i=2}^{n} e_i$ are injective edges.*

Injective circuits and broadcast paths in a data-flow 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 I/O lower bounds, while the use of broadcast paths for lower bound analysis is explained in another example that follows.

Injective circuits: In the Jacobi example, we have three circuits to vertex $S2$ through $S3$. 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 $c_1 = (e7, e8)$:
  R_{c_1} = `[T,N] -> {S2[t,i]->S2[t+1,i+1] : 1<=t<T-1 and`

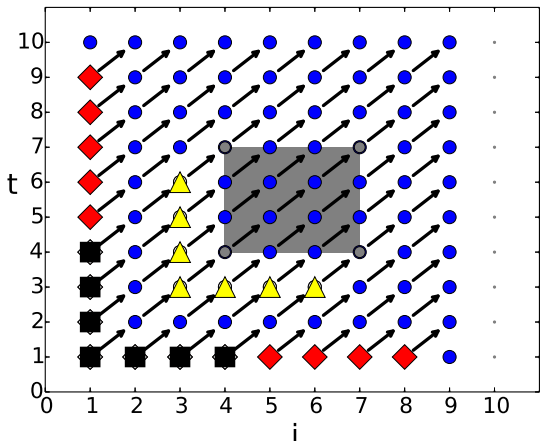


Figure 7: Original iteration domain space for Jacobi 1D. Blue circles: Integer points of domain D_{S2}; Black arrows: Relation R_{c1} of circuit $(e7,e8)$; Red diamonds: Frontier F_1; Gray box: subset E and corresponding vertex-set υ_1; Yellow triangles: $\mathsf{In}(\upsilon_1)$; Black squares: Projection of the points inside gray box along the direction of black arrows onto the frontier.

```
1<=i<N-2}
```
$\vec{b_1} = (1,1)^T$

- Circuit $c_2 = (e7, e9)$:
  R_{c_2} = `[T,N] -> {S2[t,i]->S2[t+1,i] : 1<=t<T-1 and 1<=i<N-1}`
  $\vec{b_2} = (1,0)^T$
- Circuit $c_3 = (e7, e10)$:
  R_{c_3} = `[T,N] -> {S2[t,i]->S2[t+1,i-1] : 1<=t<T-1 and 2<=i<N-1}`
  $\vec{b_3} = (1,-1)^T$

Fig. 7 pictorially shows the domain D_{S2} and the relation R_{c_1} as a $\mathcal{Z}$-polyhedron for $T = N = 11$.

DEFINITION 8 (Frontier). *The frontier, F, of a relation R, with domain D, is the set of points with no incoming edges in the corresponding $\mathcal{Z}$-polyhedron.*

F can be calculated using the set operation $F = D \setminus R(D)$. The frontiers F_1, F_2 and F_3 for the three relations, R_{c_1}, R_{c_2} and R_{c_3} respectively, are listed below.

- F_1 = `[T,N]->{S2[1,i]:2<=i<N-2; S2[t,1]:1<=t<T-1}`
- F_2 = `[T,N]->{S2[1,i]:1<=i<N-1}`
- F_3 = `[T,N]->{S2[1,i]:2<=i<N-1; S2[t,N-2]:2<=t<T-1}`

In Fig. 7, points of frontier F_1 are shown as red diamond shaped points. Due to the correspondence between a $\mathcal{Z}$-polyhedron and a (sub-)CDAG (refer to Sec. 5.1), each point in a frontier represents a source vertex (i.e., vertex with no incoming edges) of the (sub-)CDAG. It could be seen that there are $|F_i|, 1 \leq i \leq 3$, disjoint paths $P_i, 1 \leq i \leq 3$ (as a consequence of the injective property of the relations) in the sub-CDAG $C_1 = (I_1, V_1, E_1, O_1)$ (corresponding to the instances of statements $S2$ and $S3$), each with a distinct source vertex that corresponds to a point in F_i. These source vertices are tagged as inputs for the lower bounds analysis and their count, $|F_i|$, is later subtracted from the final I/O lower bound using Theorem 4.

Let $\upsilon_1 \subseteq V_1$ be a vertex-set of a valid $2S$-partition of C_1. There are a set of points E in the $\mathcal{Z}$-polyhedron (e.g., the set of points inside the gray colored box in Fig. 7) corresponding to υ_1. The set of points outside E with an edge to a point in E corresponds to $\mathsf{In}(\upsilon_1)$ (e.g., points marked with yellow triangles in Fig. 7). Since there is no cyclic dependence between the vertex-sets of the $2S$-partition and the paths are disjoint, by starting from the vertices of $\mathsf{In}(\upsilon_1)$ and tracing backwards along the paths in any P_i, $1 \leq i \leq 3$, we should reach $|\mathsf{In}(\upsilon_1)| \leq 2S$ distinct source vertices. This process corresponds to projecting the set E along each of the directions $\vec{b_i}$, $1 \leq i \leq 3$ onto the frontier F_i, $1 \leq i \leq 3$. Hence we have $|E_{\downarrow \vec{b_i}}| \leq 2S$ (here, $E_{\downarrow \vec{b_i}}$ denotes projecting E along the direction $\vec{b_i}$). The points of the frontier obtained by projection are shown as black squares (over red diamonds) in Fig. 7. We ensure that $E \subseteq D^R_{S2} = \mathsf{domain}(R_{c_1}) \cap \mathsf{image}(R_{c_1}) \cap \mathsf{domain}(R_{c_2}) \cap \mathsf{image}(R_{c_2}) \cap \mathsf{domain}(R_{c_3}) \cap \mathsf{image}(R_{c_3})$. This allows us to apply the geometric reasoning discussed in Sec. 5.2 to restrict the size of the set E as shown below. Since $\dim(D_{S2}) = 2$, it is sufficient to consider any two linearly independent directions.

Theorem 6 applies only for projections along the orthogonal directions. In case projection vectors are non-orthogonal, a simple change of basis operation is used to transform the space to a new space where the projection directions are the canonical bases. In the example, if we consider vectors $\vec{b_1}/|\vec{b_1}|$ and $\vec{b_3}/|\vec{b_3}|$ as the projection directions in the original space, then the linear map $\begin{pmatrix} \vec{b_1}/|\vec{b_1}| & \vec{b_3}/|\vec{b_3}| \end{pmatrix}^{-1}$ will transform the $\mathcal{Z}$-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 projections: $\phi_1 : (i,j) \rightarrow (i)$; $\phi_2 : (i,j) \rightarrow (j)$. From Eq. 6, we obtain the following inequalities for the dual problem (refer (8)): $x_1 \leq 1$; $x_2 \leq 1$.

In addition, we also need to 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: $|\phi_1(E)|^{x_1} \leq (N+T)$; $|\phi_2(E)|^{x_2} \leq (N+T)$, or (after taking log with base S), $x_1 \log_S(|\phi_1(E)|) \leq \log_S(N+T)$; $x_2 \log_S(|\phi_1(E)|)) \leq \log_S(N+T)$. Since $|\phi_j(E)| \leq S$, we have $\log_S(|\phi_j(E)|) \leq 1$. 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.

$$\text{Maximize } \Theta = x_1 + x_2 \tag{10}$$

$$\begin{aligned} \text{s.t.} \quad x_1 &\leq 1 \\ x_2 &\leq 1 \\ x_1 &\leq \log_S(N+T) \\ x_2 &\leq \log_S(N+T) \end{aligned}$$

Solving Eq. (10) using PIP [16] provides the following solution: If $\log_S(N+T) \geq 1$ then, $x_1 = x_2 = 1$, else, $x_1 = x_2 = \log_S(N+T)$.

This specifies that when $N+T = \Omega(S)$, $|\upsilon_1| = O(S^2)$, and hence $Q = \Omega\left(\frac{NT}{S} - (N+T)\right)$ (here, $(N+T)$ is subtracted from the lower bound to account for I/O tagging), otherwise, $|\upsilon_1| = O\left((N+T)^2\right)$ and $Q = \Omega\left(\frac{NTS}{(N+T)^2} - (N+T)\right)$.

In the example, since the vectors $\vec{b_1}/|\vec{b_1}|$ and $\vec{b_3}/|\vec{b_3}|$ are already orthonormal, the change of basis transformation that we performed earlier is unimodular. But, in general this need not be the case. Since we focus only on asymptotic parametric bounds, any constant multiplicative factors that arise due to the non-unimodular transformation are ignored.

Illustrative example 2: The following example is composed of a scaled matrix-multiplication and a Gauss-Seidel computation within an outer iteration loop.

```
Parameters: W, N, T
Inputs: A[N][N], C[N][N], Temp[N][N]
Outputs: A[N][N], C[N][N]
   // Iterative loop with scaled Matmult
```

```
    // followed by Stencil
    for(it=0;it<W;it++)
    {
      // Scaled Matmult split out into a sequence of
      // mat-vec and vector scaling ops for each row
      for(i=0;i<N;i++)
      {
        for(j=0;j<N;j++)
          for(k=0;k<N;k++)
S1:         Temp[i][j] += A[i][k]*A[k][j];

        for(j=0;j<N;j++)
S2:       Temp[i][j] = 2*Temp[i][j];

        for(j=0;j<N;j++)
S3:       C[i][j] += Temp[i][j];
      }

      // Seidel stencil
      for(t=0;t<T;t++)
        for (i=1; i<N-1; i++)
          for (j=1; j<N-1; j++)
S4:         A[i][j] = 0.5 * (A[i-1][j] + A[i][j-1] \
              + A[i][j] + A[i+1][j] + A[i][j+1]);
    }
```

The decomposition theorem (Theorem 2) allows us to split this code into individual components, analyze each sub-program separately and obtain the I/O lower bounds for the whole program through simple summation of the individual bounds. Hence, given the CDAG C of the above example, the analysis proceeds with the following steps:

- The CDAG C and thus the underlying program is decomposed as follows: (1) Each iteration of the outer loop, with trip-count `W`, is split into W sub-programs. (2) Each of this sub-program is further decomposed by separating the matmult (consisting of statements $S1$, $S2$ and $S3$) and Seidel operations (consisting of statement $S4$) into individual sub-programs.
- The vertices corresponding to the input arrays of the matmult and Seidel computations are tagged as inputs in their corresponding sub-CDAGs.
- The matmult (with sub-CDAG $C_m = (I_m, V_m, E_m, O_m)$) and the Seidel computation (with sub-CDAG $C_s = (I_s, V_s, E_s, O_s)$) are separately analyzed for their I/O lower bounds.
- If L_m and L_s are the I/O lower bounds obtained in the previous step for matmult and Seidel computation, respectively, Theorem 2 and Theorem 4 provides us an I/O lower bound of $\Omega(W \times ((L_m - |I_m|) + (L_s - |I_s|)))$ for the whole program.

The analysis of the Seidel computation is similar to the analysis of the Jacobi 1D computation detailed in the previous example. Hence, we skip the analysis and provide the following result: If $N = \Omega(\sqrt{S})$ and $T = \Omega(\sqrt{S})$ then, $Q_s = \Omega\left(\frac{N^2T}{\sqrt{S}} - N^2 - NT\right)$, else $Q_s \geq 0$. where, Q_s is the I/O complexity for the Seidel computation.

Now, we consider the analysis of the scaled matmult. The data-flow graph, G_F consists of six vertices: vertices A, C and $Temp$ correspond to the input arrays `A`, `C` and `Temp`, respectively; vertices $S1$, $S2$ and $S3$ correspond to the statements $S1$, $S2$ and $S3$, respectively. The domain corresponding to each vertex (in the order A, C, $Temp$, $S1$, $S2$ and $S3$) is listed below:

- D_A = `[N]->{A[i,j]:0<=i<N and 0<=j<N}`
- D_C = `[N]->{C[i,j]:0<=i<N and 0<=j<N}`
- D_{Temp} = `[N]->{Temp[i,j]:0<=i<N and 0<=j<N}`
- D_{S1} = `[N]->{S1[i,j,k]:0<=i<N and 0<=j<N and 0<=k<N}`
- D_{S2} = `[N]->{S2[i,j]:0<=i<N and 0<=j<N}`
- D_{S3} = `[N]->{S3[i,j]:0<=i<N and 0<=j<N}`

The relations corresponding to various edges are listed below.

- $\mathsf{relation}(e1 = (A, S1)) = R_{e1}$ = `[N] -> {A[i, j] -> S1[i,j',j] : 0<=i<N and 0<=j<N and 0<=j'<N}`
- $\mathsf{relation}(e2 = (A, S1)) = R_{e2}$ = `[N] -> {A[i,j] -> S1[i',j,i] : 0<=i'<N and 0<=i<N and 0<=j<N}`
- $\mathsf{relation}(e3 = (C, S3)) = R_{e3}$ = `[N] -> {C[i,j] -> S3[i,j] : 0<=i<N and 0<=j<N}`
- $\mathsf{relation}(e4 = (Temp, S1)) = R_{e4}$ = `[N] -> {Temp[i,j] -> S1[i,j,0] : 0<=i<N and 0<=j<N}`
- $\mathsf{relation}(e5 = (S1, S1)) = R_{e5}$ = `[N] -> {S1[i,j,k] -> S1[i,j,k+1] : 0<=i<N and 0<=j<N and 0<=k<N-1}`
- $\mathsf{relation}(e6 = (S1, S2)) = R_{e6}$ = `[N] -> {S1[i,j,N-1] -> S2[i,j] : 0<=i<N and 0<=j<N}`
- $\mathsf{relation}(e7 = (S2, S3)) = R_{e7}$ = `[N] -> {S2[i,j] -> S3[i,j] : 0<=i<N and 0<=j<N}`

Broadcast paths: The paths $p_1 = (e1)$ and $p_2 = (e2)$ are of type broadcast. As p_1 and p_2 are composed of a single edge, their relations, R_{p_1} and R_{p_2} respectively, are the same as their edge. Thus, $R_{p_1} = R_{e1}$ and $R_{p_2} = R_{e2}$. We are specifically interested in the broadcast paths whose inverse-relations (e.g., $R_{p_1}^{-1}$) can be expressed as affine maps. In our example, the two inverse-relations $R_{p_1}^{-1}$ and $R_{p_2}^{-1}$ can be expressed as affine maps as shown below:

$$R_{p_1}^{-1} \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} i \\ j' \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

$$R_{p_2}^{-1} \equiv \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} \cdot \begin{pmatrix} i' \\ j \\ i \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Further, we have an injective circuit $p_3 = (e5)$ with $R_{p_3} = R_{e5}$, whose direction vector $\vec{b_3} = (0,0,1)^T$.

We next calculate the frontiers F_1, F_2 and F_3 of the relations R_{p_1}, R_{p_2} and R_{p_3}, respectively, by taking the set-difference of their domain and image (e.g., $F_1 = D_{p_1} \setminus R_{p_1}(D_{p_1})$), where, $D_{p_1} = \mathsf{domain}(p_1)$). The three frontiers are shown using the ISL notation below:

- F_1 = `[N] -> {A[i,j] : 0<=i<N and 0<=j<N}`
- F_2 = `[N] -> {A[i,j] : 0<=i<N and 0<=j<N}`
- F_3 = `[N] -> {S1[i,j,0] : 0<=i<N and 0<=j<N}`

In the case an injective circuit (with associated relation, say, R_a), we chose the direction of projection to be the vector representing R_a. Here, in case of a broadcast path (with associated relation, say, $R_b = \mathbf{A}.\vec{x} + \vec{b}$), we choose the kernel of the matrix $\mathbf{A}$, $\ker(\mathbf{A})$, to be the projection direction. The intuition behind choosing this direction is that the kernel represents the plane of reuse, and hence, the set of points obtained by projecting a set E along the kernel directions represents the $\mathsf{In}(E)$. In general, the kernel can be of dimension higher than one (but has to be at least one due to the definition of a broadcast path). The kernels ($\vec{k_1}$ and $\vec{k_2}$) of the inverse-relations of the paths p_1 and p_2 are: $\vec{k_1} = (0,1,0)^T$ and $\vec{k_2} = (1,0,0)^T$, respectively.

By choosing $\vec{k_1}$, $\vec{k_2}$ and $\vec{b_3}$ as the projection directions, we obtain $\phi_1 : (i,j,k) \to (i,k)$; $\phi_2 : (i,j,k) \to (j,k)$; $\phi_3 : (i,j,k) \to (i,j)$. 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 the degenerated cases, we have the additional constraints that specify that the size of the projections onto the subspaces $\{\vec{i}\}$, $\{\vec{j}\}$ and $\{\vec{k}\}$ should not exceed N and the size of the projections onto the subspaces $\{\vec{i},\vec{j}\}$, $\{\vec{j},\vec{k}\}$ and $\{\vec{i},\vec{k}\}$ should not exceed N^2. Hence, we obtain the following parametric linear programming problem.

$$\text{Maximize } \Theta = x_1 + x_2 + x_3 \tag{11}$$

$$
\begin{array}{rrcl}
\text{s.t.} & x_1 + x_2 & \leq & 1 \\
& x_2 + x_3 & \leq & 1 \\
& x_1 + x_3 & \leq & 1 \\
& x_1 & \leq & \log_S(N) \\
& x_2 & \leq & \log_S(N) \\
& x_3 & \leq & \log_S(N) \\
& x_1 + x_2 & \leq & 2\log_S(N) \\
& x_2 + x_3 & \leq & 2\log_S(N) \\
& x_1 + x_3 & \leq & 2\log_S(N)
\end{array}
$$

Solving Eq. (11) using PIP [16] provides 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, when $N = \Omega(\sqrt{S})$, $Q_m = \Omega\left(\frac{N^3}{\sqrt{S}} - N^2\right)$, otherwise, $Q_m \geq 0$.

Finally, by applying Theorem 2, we obtain the I/O lower bound for the full program, $Q \geq Q_m + Q_s = \Omega\left(W \times \left(\frac{N^3}{\sqrt{S}} + \frac{N^2 T}{\sqrt{S}} - N^2 - NT\right)\right)$ when N and T are sufficiently large.

Putting it all together: Algorithm 1 provides a pseudo-code for our algorithm. Because 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 (the higher the kernel, the more the reuse, the 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. The role of the function try() (on lines 20, 26 and 32 in Algorithm 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 funtion try() is shown in Algorithm 3. The function best(v) (shown in Algorithm 2) selects a set of paths for a vertex v and computes the associated complexity. The function solve() (shown in Algorithm 4) writes the linear program and returns the I/O 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, $\mathsf{domain}(R)$ and $\mathsf{image}(R)$ return the domain and image of R, respectively.
- For an edge e, the operation $\mathsf{relation}(e)$ provides its associated relation. If $R_e = \mathsf{relation}(e)$ 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., $\mathsf{relation}(p)$ 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. $\mathsf{domain}(R_i \circ R_j) = R_j^{-1}(\mathsf{image}(R_j) \cap \mathsf{domain}(R_i))$ and $\mathsf{image}(R_i \circ R_j) = R_i(\mathsf{image}(R_j) \cap \mathsf{domain}(R_i))$.
- For a given domain D, $\dim(D)$ 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 $\mathcal{B}$), and computing $\log_B(|D|)$, and rounding the result to the nearest integer. For example, if $|D| = C(n,m) = nm + n + 3$, setting $\mathcal{B} = 10^3$, we get $\dim(D) = \mathsf{round}\left(\log_{\mathcal{B}}(C(\mathcal{B},\mathcal{B}))\right) = 2$.
- If a relation R is injective and can be expressed as an affine map of the form $\mathbf{A}.\vec{x} + \vec{b}$, then the operation $\mathsf{ray}(R)$ computes $\vec{b}$, otherwise, returns $\perp$.
- For a relation R, if its inverse can be expressed as an affine relation $\mathbf{A}.\vec{x} + \vec{b}$, $\mathsf{rkernel}(R)$ computes the kernel of the matrix A (and returns $\perp$ otherwise).
- For a set of vectors $b = \{\vec{b_1}, \ldots, \vec{b_l}\}$, $\mathsf{subspace}(b)$ provides the linear subspace spanned by those vectors.
- For a set of linear subspaces $K = \{k_1, \ldots, k_l\}$, $\mathsf{base}(K)$ gives a set of linearly independent vectors $b = \{\vec{b_1}, \ldots, \vec{b_d}\}$ such that for any k_i, there exists $b_i \subseteq b$ s.t. $k_i = \mathsf{subspace}(b_i)$. If such a set could not be computed, it returns $\perp$.
- For a path p, $\mathsf{vertices}(p)$ returns its set of vertices.
- Given an expression X, the operation $\mathsf{simplify}(X)$ simplifies the expression by eliminating the lower order terms. For example, $\mathsf{simplify}(NT + N^2 - N + T)$ returns $NT + N^2$.

```
1  G_F = (V_F, E_F);
2  F_I := ∅; F_B := ∅; F_BB := ∅;
3  foreach e = (u,v) ∈ E_F do
4      R := relation(e);
5      if dim(image(R)) < dim(v) then next;
6      if R is invertible then F_I := F ∪ {e};
7      if dim(domain(R)) = dim(image(R)) − 1 then
8          F_B := F_B ∪ {e}
9      end
10     if dim(domain(R)) < dim(image(R)) − 1 then
11         F_BB := F_BB ∪ {e}
12     end
13 end
14 foreach v ∈ V_F do
15     d := dim(v);
16     foreach circuit p from v to v in F_I do
17         R := relation(p);
18         if (b := ray(R)) = ⊥ then next p;
19         if dim(image(R)) < d then next p;
20         if try(v, subspace(b), p) then next v;
21     end
22     foreach cycle-free path p to v in F_B F_I^* do
23         R := relation(p);
24         if (k := rkernel(R)) = ⊥ then next p;
25         if dim(image(R)) < d then next p;
26         if try(v,k,p) then next v;
27     end
28     foreach cycle-free path p to v in F_BB F_I^* do
29         R := relation(p);
30         if (k := rkernel(R)) = ⊥ then next p;
31         if dim(image(R) < d) then next p;
32         if try(v,k,p) then next v;
33     end
34     best(v);
35 end
36 simplify(∑_{v∈V_F} v.complexity);
```

Algorithm 1: For each vertex v in a data-flow graph G_F, finds a set of paths and computes the corresponding complexity.

```
1 Function best(vertex v)
2 let (k,K,D,T) ∈ v.clique with maximum lexicographic value of
    (dim(D), dimension(k), −∑_{k_i∈K} dimension(k_i), solve(D,K), −|T|);
3 Q := solve(D,K);
4 v.complexity := Q;
```

Algorithm 2: For a vertex v, selects a set of paths and computes the associated complexity.

```
1  Function try(vertex v, subspace k', path p)
2  {v.clique is a set of quadruples (k,K,D,T) where:
       - k is a subspace,
       - K is a set of subspaces,
       - D is a domain,
       - T is a set of vertices}
3  {v.complexity is an asymptotic complexity (with cases)}
4  if k' ∈ v.subspaces then return false;
5  v.subspaces := v.subspaces ∪ {k'};
6  foreach (k,K,D,T) ∈ v.clique ∪ (⊥,⊥,⊥,⊥) do
7      if dimension(k+k') > dimension(k) and
       base(K ∪ {k'}) ≠ ⊥ then
8          D' = image(relation(p)) ∩ D;
9          if D = ⊥ or dim(D') = dim(D) then
10             T' := T ∪ vertices(p);
11             K' := K ∪ {k'};
12             v.clique := v.clique ⋃ (k+k', K', D', T');
13             if dimension(k+k') ≥ dim(domain(v)) then
14                 Q := solve(D', K');
15                 v.complexity := Q;
16                 return true;
17             end
18         end
19     end
20 end
21 return false;
```

Algorithm 3: For a vertex v, try to add path p to some other paths. Return true if a good bound is found.

```
1  Function solve(domain D, set of subspaces K)
2  b := base(K);
3  LP := objective(maximize Θ = Σ_{b_i ∈ b} α_i);
4  foreach k ∈ K do LP := LP.constraint(Σ_{b_i ∉ k} α_i ≤ 1);
5  foreach b' ⊂ b do
6      D_{b'} := projection(subspace(b'), D);
7      LP := LP.constraint(Σ_{b_i ∈ b'} α_i ≤ log_S(|D_{b'}|));
8  end
9  F := Σ_{k ∈ K} |projection(¬k, D)|;
10 Θ := solution(LP);
11 U := S^Θ;
12 Q := Ω(|D|S / U) − Ω(F);
13 return Q;
```

Algorithm 4: For a domain D and a set of compatible subspaces, writes the linear program, and returns the I/O lower bound (with cases).

6. Related Work

Hong & Kung provided the first formalization of the I/O complexity problem for a two-level memory hierarchy using the red/blue pebble game on a CDAG and the equivalence to 2S-partitions of the CDAG. We perform an adaptation of Hong & Kung 2S-partitioning to constrain the size of the input set of each vertex set rather than a dominator set, which is suitable for bounding the minimum I/O for a CDAG with the restricted red/blue pebble game where repebbling is disallowed. This adaptation enables effective composition of lower bounds of sub-CDAGs to form I/O lower bounds for composite CDAGs. A similar adaptation has previously been used by modifying the red/blue pebble game through addition of a third kind of pebble [14, 15]. The composition of lower bounds for sequences of linear algebra operations has previously been addressed by the work of Ballard et al. [1] by use of "imposed" reads and writes in between segments of operations, adding the lower bounds on data access for each of the segments, and subtracting the number of imposed reads and writes. Our use of tagged inputs and outputs in conjunction with application of the decomposition theorem bears similarities to the use of imposed reads and writes by Ballard et al., but is applicable to the more general model of CDAGs that model data dependences among operations.

Bilardi et al. [8] proposed an approach to obtain lower bounds on the access complexity of a DAG in terms of space lower bounds that apply to disjoint components of the DAG, when recomputation is not allowed. The approach was later extended to the case when recomputation is allowed, by means of the notion of free-input space complexity [9].

Irony et al. [21] used a geometric reasoning with the Loomis-Whitney inequality [23] to present an alternate proof to Hong and Kung's [20] for I/O lower bounds on standard matrix multiplication. More recently, Demmel's group at UC Berkeley has developed lower bounds as well as optimal algorithms for several linear algebra computations including QR and LU decomposition and the all-pairs shortest paths problem [1, 3, 13, 33].

Extending the scope of the Hong & Kung model to more complex memory hierarchies has also been the subject of research. Savage provided an extension together with results for some classes of computations that were considered by Hong & Kung, providing optimal lower bounds for I/O with memory hierarchies [27]. Valiant proposed a hierarchical computational model [35] that offers the possibility to reason in an arbitrarily complex parametrized memory hierarchy model. While we use a single-level memory model in this paper, the work can be extended in a straight forward manner to model multi-level memory hierarchies.

Unlike Hong & Kung's original model, several models have been proposed that do not allow recomputation of values (also referred to as "no repebbling") [1, 3, 7, 10, 12, 21, 24–28, 30, 31]. Savage [27] developed results for FFT using no repebbling. Bilardi and Peserico [7] explore the possibility of coding a given algorithm so that it is efficiently portable across machines with different hierarchical memory systems, without the use of recomputation. Ballard et al. [1, 3] assume no recomputation in deriving lower bounds for linear algebra computations. Ranjan et al. [25] develop better bounds than Hong & Kung for FFT using a specialized technique adapted for FFT-style computations on memory hierarchies. Ranjan et al. [26] derive lower bounds for pebbling r-pyramids under the assumption that there is no recomputation. As discussed earlier, we also use a model that disallows recomputation of values. But our focus in this regard is different from previous efforts – we formalize an adaptation of the the 2S-partitioning model of Hong & Kung that facilitates effective composition of lower bounds from sub-CDAGs of a composite CDAG.

```
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];
```

(a) Matrix Multiplication Code

```
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;
  }
```

(b) Code with same array accesses as Mat-Mult

Figure 8: Example illustrating difference between CDAG model and computational model used by Christ et al. [11]

The previously described efforts on I/O lower bounds have involved manual analysis of algorithms to derive the bounds. In contrast, in this paper we develop an approach to automate the analysis of I/O lower bounds for programs. The only other such effort to our knowledge is the recent work of Christ et al. [11]. Indeed, the approach we have develop in this paper was inspired by their work, but differs in a number of significant ways:

1. 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. [11] are based on a different abstraction of an indivisible loop body of affine statements within a perfectly nested loop. For example, under that model, the lower bounds for codes in Fig. 8(a) (standard matrix multiplication) and Fig. 8(b) would be exactly the same – $O(N^3/\sqrt{S})$ – 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 Fig. 8(a) representing a connected CDAG, while the code in Fig. 8(b) represents has a CDAG with three disconnected parts corresponding to the three statements, and computation has a much lower I/O complexity of $O(N^2)$.
2. The work of Christ et al. [11] does not model data dependencies between statement instances, and can therefore produce weak lower bounds. In contrast, the approach developed in this paper is based on using precise data dependence information as the basis for geometric reasoning in the iteration space to derive the I/O lower bounds. For example, with the 2D-Jacobi example discussed earlier, the lower bound obtained by the approach of Christ et al. would be $O(N^2)$ instead of the tight bound of $O(N^2T/\sqrt{S})$ that is obtained with the algorithm developed in this paper.
3. This work addresses a more general model of programs. While the work of Christ et al. [11] only models perfectly nested loop computations, the algorithms presented in this paper handle sequences of imperfectly nested loop computations.

7. Discussion

We conclude by raising some issues and open questions, some of which are being addressed in ongoing work.

Tightness of lower bounds: A very important question is whether a lower bound is tight – clearly, zero is a valid but weak and useless I/O lower bound for any CDAG. The primary means of assessing tightness of lower bounds is by comparison with upper bounds from algorithm implementations that have been optimized for data locality. For example, tiling (or blocking) is a commonly used approach to enhance data locality of nested loop computations. An open question is whether any automatic tool can be designed to systematically explore the space of valid schedules to generate good parametric upper bounds based on models and/or heuristics that minimize data movement cost.

Lower bounds when recomputation is allowed: 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 I/O lower bounds under the original model of Hong & Kung that permits re-computation 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 I/O 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 (that prohibits re-pebbling) 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?*

Relating I/O lower bounds to machine parameters: I/O lower bounds can be used to determine whether an algorithm will be inherently limited to performance far below a processor's peak because of data movement bottlenecks. The collective bandwidth between main memory and the last level cache in multicore processors in words/second on current systems is over an order of magnitude lower than the aggregate computational performance of the processor cores in floating-point operations per second; this ratio is a critical machine balance parameter. By comparing this machine balance parameter to the ratio of the I/O lower bound (calculated for S set to the capacity of the last level on-chip cache) to the total number of arithmetic operations in the computation, we can determine if the algorithm will be inherently limited by data movement overheads. However, such an analysis will also require tight assessment of the constants for the leading terms in the asymptotic expressions of the order complexity for I/O lower bounds. This is not addressed by the approach presented in this paper.

Modeling associative operators: Reductions using associative operators like addition occur frequently in computations. With the CDAG model, some fixed order of execution is enforced for such computations, resulting in an over-constrained linear chain of dependencies between the vertices corresponding to instances of an associative operator. Some previously developed geometric approaches to modeling I/O lower bounds [1, 11, 21] have developed I/O lower bounds for a family of algorithms that differ in the order of execution of associative operations. It would be of interest to extend the automated lower bounding approach of this paper to also model lower bounds among a family of CDAGs corresponding to associative reordering of the operations.

Finding good decompositions: The second illustrative example in Sec. 5 demonstrated the benefit of judiciously decomposing CDAGs to obtain good lower bounds by combining bounds for sub-CDAGs via the decomposition theorem. However, if the decomposition is performed poorly, the result will be a very weak lower bound. In the same example, if the computation within the second level `i` loop had also been used to further decompose the CDAG, we would have a sequence of matrix-vector multiplications with order complexity $O(N^2)$ from which the tagged I/O nodes of the same order of complexity must be subtrated out, resulting in a weak lower bound of zero. Conversely, if the computation within the outer `it` loop were not decomposed into sub-CDAGs, it would again have resulted in weak lower bounds. The question of automatically finding effective decompositions of CDAGs to enable tight lower bounds is an open problem.

8. Conclusion

Characterizing the I/O complexity of a program is a cornerstone problem, that is particularly important with current and emerging power-constrained architectures where data movement costs are the dominant energy bottleneck. Previous approaches to modeling the I/O complexity of computations have several limitations that this paper has addressed. First, by suitably modifying the pebble game model used for characterizing I/O complexity, analysis of large composite computational DAGs is enabled by decomposition into smaller sub-DAGs, a key requirement to allow the analysis of complex programs. Second, a static analysis approach has been developed to compute I/O lower bounds, by generating asymptotic parametric data-access lower bounds for programs as a function of cache size and problem size.

Acknowledgment

We thank the anonymous referees for the feedback and many suggestions that helped us significantly in improving the presentation of the work. We thank Gianfranco Bilardi, Jim Demmel, and Nick Knight for discussions on many aspects of lower bounds modeling and their suggestions for improving the paper. This work was supported in part by the U.S. National Science Foundation through awards 0811457, 0926127, 0926687 and 1059417, by the U.S. Department of Energy through award DE-SC0012489, and by Louisiana State University.

References

[1] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Analysis Applications*, 32(3):866–901, 2011.

[2] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Brief announcement: Strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds. In *Proc. SPAA '12*, pages 77–79, 2012.

[3] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Graph expansion and communication costs of fast matrix multiplication. *J. ACM*, 59(6): 32, 2012.

[4] A. Barvinok. Computing the Ehrhart polynomial of a convex lattice polytope. *Discrete and Computational Geometry*, 12:35–48, 1994.

[5] J. Bennett, A. Carbery, M. Christ, and T. Tao. Finite bounds for Holder-Brascamp-Lieb multilinear inequalities. *Mathematical Research Letters*, 55(4):647–666, 2010.

[6] K. Bergman, S. Borkar, et al. Exascale computing study: Technology challenges in achieving exascale systems. *DARPA IPTO, Tech. Rep*, 2008.

[7] G. Bilardi and E. Peserico. A characterization of temporal locality and its portability across memory hierarchies. *Automata, Languages and Programming*, pages 128–139, 2001.

[8] G. Bilardi and F. P. Preparata. Processor - Time Tradeoffs under Bounded-Speed Message Propagation: Part II, Lower Bounds. *Theory Comput. Syst.*, 32(5):531–559, 1999.

[9] G. Bilardi, A. Pietracaprina, and P. D'Alberto. On the space and access complexity of computation DAGs. In *Graph-Theoretic Concepts in Computer Science*, volume 1928 of *LNCS*, pages 81–92. 2000.

[10] G. Bilardi, M. Scquizzato, and F. Silvestri. A lower bound technique for communication on bsp with application to the fft. In *Euro-Par*, pages 676–687, 2012.

[11] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. Yelick. Communication Lower Bounds and Optimal Algorithms for Programs That Reference Arrays Part 1. EECS Technical Report EECS–2013-61, UC Berkeley, May 2013.

[12] S. A. Cook. An observation on time-storage trade off. *J. Comput. Syst. Sci.*, 9(3):308–316, 1974.

[13] J. Demmel, L. Grigori, M. Hoemmen, and J. Langou. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Scientific Computing*, 34(1), 2012.

[14] V. Elango, F. Rastello, L.-N. Pouchet, J. Ramanujam, and P. Sadayappan. Data access complexity: The red/blue pebble game revisited. Technical report, OSU/INRIA/LSU/UCLA, Sept. 2013. OSU-CISRC-7/13-TR16.

[15] V. Elango, F. Rastello, L. Pouchet, J. Ramanujam, and P. Sadayappan. On characterizing the data movement complexity of computational dags for parallel execution. In *26th ACM Symposium on Parallelism in Algorithms and Architectures, SPAA '14*, pages 296–306, 2014.

[16] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[17] S. H. Fuller and L. I. Millett. *The Future of Computing Performance: Game Over or Next Level?* The National Academies Press, 2011.

[18] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Intl. J. of Parallel Programming*, 34(3), 2006.

[19] G. Gupta and S. Rajopadhye. The Z-polyhedral model. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*, pages 237–248. ACM, 2007.

[20] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. of the 13th annual ACM sympo. on Theory of computing (STOC'81)*, pages 326–333. ACM, 1981.

[21] D. Irony, S. Toledo, and A. Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, 2004.

[22] A. Ketterlin and P. Clauss. Profiling data-dependence to assist parallelization: Framework, scope, and optimization. In *MICRO*, pages 437–448, 2012.

[23] L. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bull. Am. Math. Soc.*, 55:961–962, 1949.

[24] D. Ranjan and M. Zubair. Vertex isoperimetric parameter of a computation graph. *Int. J. Found. Comput. Sci.*, 23(4):941–964, 2012.

[25] D. Ranjan, J. Savage, and M. Zubair. Strong I/O lower bounds for binomial and FFT computation graphs. In *Computing and Combinatorics*, volume 6842 of *LNCS*, pages 134–145. Springer, 2011.

[26] D. Ranjan, J. E. Savage, and M. Zubair. Upper and lower I/O bounds for pebbling r-pyramids. *J. Discrete Algorithms*, 14:2–12, 2012.

[27] J. Savage. Extending the Hong-Kung model to memory hierarchies. In *Computing and Combinatorics*, volume 959 of *LNCS*, pages 270–281. 1995.

[28] J. E. Savage. *Models of Computation*. Addison-Wesley, 1998.

[29] J. E. Savage and M. Zubair. A unified model for multicore architectures. In *Proceedings of the 1st International Forum on Next-generation Multicore/Manycore Technologies*, page 9. ACM, 2008.

[30] J. E. Savage and M. Zubair. Cache-optimal algorithms for option pricing. *ACM Trans. Math. Softw.*, 37(1), 2010.

[31] M. Scquizzato and F. Silvestri. Communication lower bounds for distributed-memory computations. *CoRR*, abs/1307.1805, 2013.

[32] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25, 2011.

[33] E. Solomonik, A. Buluç, and J. Demmel. Minimizing communication in all-pairs shortest paths. In *IPDPS*, 2013.

[34] S. I. Valdimarsson. The Brascamp-Lieb polyhedron. *Can. J. Math.*, 62(4):870–888, 2010.

[35] L. G. Valiant. A bridging model for multi-core computing. *J. Comput. Syst. Sci.*, 77:154–166, Jan. 2011.

[36] S. Verdoolaege. isl: An integer set library for the polyhedral model. In *Mathematical Software–ICMS 2010*, pages 299–302. Springer, 2010.