Dataflow-Specific Algorithms for Resource-Constrained Scheduling and Memory Design

Abhishek Bhattacharjee
Yale University
abhishek@cs.yale.edu

Quanquan C. Liu
Yale University
quanquan.liu@yale.edu

Rajit Manohar
Yale University
rajit.manohar@yale.edu

Raghavendra Pradyumna Pothukuchi
Yale University
raghav.pothukuchi@yale.edu

Muhammed Ugur*
Yale University
muhammed.ugur@yale.edu

Abstract

We introduce the *Weighted Red-Blue Pebble Game*, an extension of the classic red-blue pebble game with weighted operation costs. This weighted formulation enables constant-factor analysis of highly resource-constrained systems with bounded fast memory, unlimited slow memory, and strict energy and power constraints.

We apply our model to computational kernels in ultra-low-power brain-computer interfaces (BCIs) implanted near the brain. We express these kernels as computational directed acyclic graphs (CDAGs), enabling modular composition of operation schedules with data movement. We derive theoretically optimal schedules for a broad class of tree-structured CDAGs and apply them to on-chip memory design with circuit-level validations for power and area.

Our algorithms result in an average 63% memory area reduction and 43% static power reduction for BCI workloads—critical improvements for ensuring safe, thermally constrained operation in implantable devices. Beyond BCIs, our results underscore the broader utility of weighted pebble games in optimizing memory and I/O across resource-constrained computing environments.

CCS Concepts

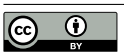
• **Theory of computation → Theory and algorithms for application domains**; • **Hardware → Application-specific VLSI designs**.

Keywords

Red-Blue Pebble Games, Data Movement, Computational Dataflow Graphs, Memory Design, Resource-Aware Algorithms

ACM Reference Format:
Abhishek Bhattacharjee, Quanquan C. Liu, Rajit Manohar, Raghavendra Pradyumna Pothukuchi, and Muhammed Ugur. 2025. Dataflow-Specific Algorithms for Resource-Constrained Scheduling and Memory Design. In *37th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '25), July 28-August 1, 2025, Portland, OR, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3694906.3743342

*Corresponding author



SPAA '25, Portland, OR, USA

ACM ISBN 979-8-4007-1258-6/2025/07
https://doi.org/10.1145/3694906.3743342

1 Introduction

Efficient scheduling of computation is a foundational challenge across computer science. Theoretically-backed scheduling strategies have traditionally been used to minimize runtime and memory usage in software. A cornerstone of these strategies is the field of *pebble games*—combinatorial models originally developed to study register allocation [37, 45], and later extended to the formal analysis of computation scheduling and I/O complexity [4, 5, 7, 12–14, 19, 23, 26, 27, 31, 32, 36, 38].

In this work, we introduce the *Weighted Red-Blue Pebble Game*, an extension of the classic red-blue pebble game [22] that incorporates weights to model cost-sensitive constraints in modern systems. While traditional pebble games offer fine-grained scheduling insights, their reliance on low-level CDAGs have limited their practical use. Coarser-grained theoretical models have emerged, but their large constant factors often make them unsuitable for highly resource-constrained domains [44]. We seek to revive pebble games to address limitations in asymptotic analysis and formalize design choices in emerging power-, energy-, and area-constrained settings.

To showcase the benefits of the weighted model, we study a system that uses mixed-memory technologies and mixed numerical precision to maximize performance under power constraints. By mixed-memory, we refer to a capacity-constrained fast memory with relatively high power consumption that is backed by a slower, larger, and more power-efficient memory. By mixed precision, we refer to the case of compute logic attached to memory which may vary in bit-width to the lowest possible value that still achieves the desired accuracy for the computational task, thereby minimizing power. These optimizations are widely used in resource-constrained domains, motivating the need for fine-grained weighted pebbling.

In the classic red-blue pebble game [22], computations are modeled as CDAGs, where each node represents either input data or a compute operation. Placement of a red pebble on a node represents performing that operation and placing its result in fast memory. A blue pebble can be placed on a node with a red pebble to perform a red-blue transition (and vice versa). These transitions represent data copies between fast and slow memories. The first placement of a red pebble on a node can only occur if all of the node's predecessors contain red pebbles; *i.e.*, all operation dependencies have been satisfied. In this classical model, all operations are assumed to have the same cost. Our Weighted Red-Blue Pebble Game maintains the same rules, but models the diverse memory costs of operations in workloads using node weights.

The objective of the classic red-blue pebble game is to minimize red-blue transitions while respecting a limit on the number of red pebbles in the graph (representing the bounded fast memory). Instead, our Weighted Red-Blue Pebble Game uses weights to represent the impact of operation heterogeneity, such as mixed precision, on memory usage, minimizing the total weighted red-blue transitions while respecting a limit on the total weight of all red pebbles in the graph at any point. This minimizes the total data transferred, and by extension, the energy cost of the schedule.

Our weighted model is designed to facilitate reusable analysis with applications to many domains through modularization. Our approach yields *optimum, exact* schedules (*i.e.*, not asymptotic schedules) for *modules* within the broad class of tree-based graphs; these modules are CDAGs that can be reused within large graphs or across graphs to perform different computational tasks. The designer can therefore express computational tasks in parts, where each part is associated with an efficient pebbling algorithm that produces minimum-cost schedules. These schedules can then be stitched together and reordered to obtain an efficient schedule for the overall computational task.

To demonstrate the power of our approach, we focus on BCIs implanted near the brain—a fast-emerging neurotechnology critical to treating severe neurological and psychiatric disorders. BCIs involve surgically implanted electrodes that read from and stimulate neurons in the brain, and have shown success in treating diseases like epilepsy, Parkinson's, chronic pain, and paralysis in multiple studies [1, 15, 18, 30, 33, 41]. Realizing the potential of BCIs rests on designing ultra-efficient computation and data movement schedules under severe power limits—often under a few milliwatts, nearly 100× less than smartphone processors [24, 41]. Implanted BCIs that even slightly increase brain temperature can induce seizures, or long-term neurological damage, making power efficiency paramount to safe implantation [25].

Our Weighted Red-Blue Pebble Game is well-suited to BCI design, as neuroengineers are exploring the design of mixed-memories (*i.e.*, fast SRAM with slow non-volatile memory such as Flash) and numerical precision to balance the quality of disease treatment with power [24, 41, 43]. Thus far, they have used ad hoc approaches to identify data movement schedules for their processors.

In this paper, we show how our weighted model can systematically help BCI design via a comprehensive experimental analysis on two key graph constructions—the Discrete Wavelet Transform (DWT) and Matrix-Vector Multiplication (MVM)—by comparing the weighted I/O cost and minimum fast memory size of our schedules against the best-known prior work of IOOpt [34, 35]. These workloads are familiar to BCI designers and capture the core signal processing and linear algebra kernels used in detection of seizures and in assessing the intended movement of paralyzed individuals [24, 41]. They are also more broadly representative of other algorithms used in BCIs; *e.g.*, DWT's recursive divide-and-conquer structure appears in filters and fast Fourier transforms, while MVM extends to classification and principal-component analysis, and builds the foundation for the broad set of tensor operations.

Our evaluations quantify improvements in metrics like data transferred and minimum fast memory size, and for hardware design, circuit power, performance, and area. We perform detailed physical synthesis and hardware modeling to accurately capture these metrics down to the circuit and wire-level. Moreover, since previous approaches did not account for mixed-precision workloads, we adapt their lower and upper bounds to preserve workload-specific characteristics for fair comparison. For the DWT graphs, we employ a layer-by-layer scheduling heuristic as an upper bound.

Overall, we demonstrate that the Weighted Red-Blue Pebble Game offers superior hardware implementation of application-specific designs. The average memory size reduction in the node-weighted case reaches 46.8% and 36.2% for the DWT and MVM, respectively. This leads to a practical on-chip area savings of 89.5% and 52.6% which reduces static power dissipation by 64.7% and 39.4% respectively compared to memory sizes given by other approaches.

In summary, our technical contributions are:

- The introduction of the Weighted Red-Blue Pebble Game to formally study the interaction between data movement on a two-level memory hierarchy and operation weights.
- Polynomial-time pebbling algorithms for generating optimum weighted schedules on specific dataflows, particularly the broad class of k-ary trees. We motivate our algorithm design with DWT and MVM tiling.
- Evaluation of our dataflow-specific algorithms in terms of data transferred and minimum fast memory size with detailed physical synthesis, improving leakage and area.

In Section 2, we describe our Weighted Red-Blue Pebble Game. In Section 3, we describe our approach to designing dataflow-specific pebbling algorithms. We extend our algorithms to include data reuse and memory states in Section 4. We evaluate DWT and MVM in Section 5. An overview of related work is given in Section 6.

2 Weighted Red-Blue Pebble Game

In this section, we describe the Weighted Red-Blue Pebble Game (WRBPG). The WRBPG introduces weighted graph vertices and a weighted red pebble constraint. This variation is played on a CDAG with the same moves, two-level memory hierarchy, and rules as the original game, namely:

- $M1$, copy to fast memory (add a red pebble to a node with a blue pebble)
- $M2$, copy to slow memory (add a blue pebble to a node with a red pebble)
- $M3$, perform a computation (if all nodes with incoming edges have a red pebble, add a red pebble)
- $M4$, delete a red pebble (blue pebbles are never deleted).

We constrain the red pebbles placed on the CDAG at any given time not by their *number* (as in the original game), but by their total weighted cost (Definition 2.1). The subsequent sections formally define the model, describe the weighted constraint and model properties, and outline how this model can be used to study efficient resource-constrained scheduling and memory design.

2.1 Model Definitions

Let $G = (V, E, w, B)$ be a node-weighted CDAG, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, $B \in \mathbb{R}_{>0}$ is the weighted red pebble budget, and $w = (w_v)_{v \in V}$ assigns a weight $w_v \in \mathbb{R}_{>0}$ to each node $v \in V$. We assume each weight w_v and the budget B

have polynomial precision, *i.e.*, they can be represented using at most poly$(|V| + |E|)$ bits.

We define the source (input) nodes of G, *i.e.*, all nodes with in-degree zero, as $\mathcal{A}(G)$, and the sink (output) nodes of G, *i.e.*, all nodes with out-degree zero, as $\mathcal{Z}(G)$. We assume $\mathcal{A}(G) \cap \mathcal{Z}(G) = \emptyset$. In the WRBPG, all $v \in \mathcal{A}(G)$ starts with a blue pebble, and the game is finished once all $v \in \mathcal{Z}(G)$ have a blue pebble. This is referred to as the starting and stopping condition respectively. We additionally define $\mathcal{H}(v) \subset V$ as the set of immediate predecessors (parents) for a node $v \in V$, *i.e.*, if $v \in \mathcal{A}(G)$, then $\mathcal{H}(v) = \emptyset$.

A red-blue pebbling of G is a sequence of moves $M1(v)$, $M2(v)$, $M3(v)$, or $M4(v)$ performed on nodes $v \in V$ that adhere to the rules defined above and fulfills the stopping condition. This sequence is denoted as $S^G = (\sigma_1, \ldots, \sigma_t)$ and is referred to as a ***schedule***.

Schedules are equivalently represented as a sequence of snapshots $(C_0, C_1, C_2, \ldots, C_t)$. A snapshot is a node-labeled graph $C_i = (V, E, \lambda)$ corresponding to each σ_i in a schedule and $\lambda = (\lambda_v)_{v \in V}$ assigns a label $\lambda_v \in \{red, blue, both, none\}$ to each node. Performing a move σ_i involves changing the label of a node in the previous snapshot C_{i-1}. Figure 1 shows the label transitions of a move and previous node label based on the rules of the red-blue pebble game. Furthermore, the snapshots C_0 and C_t correspond to the starting and stopping conditions respectively.

For a given snapshot C_i, we define $\mathcal{R}(C_i) \subseteq V$ as the nodes with $\lambda_v = red$ or $both$, $\mathcal{B}(C_i) \subseteq V$ as the nodes with $\lambda_v = blue$ or $both$, and $\mathcal{N}(C_i) \subseteq V$ as the nodes with $\lambda_v = none$. Using these definitions, the red pebble game constraint in the original game ensures that $|\mathcal{R}(C_i)| \leq R$ for some number of red pebbles $R \in \mathbb{Z}_{>0}$ and $0 \leq i \leq t$ given any red-blue pebble game schedule [22]. We modify this constraint for the weighted setting as follows:

Definition 2.1 (Weighted Red Pebble Constraint). *Let $G = (V, E, w, B)$ be a node-weighted CDAG with budget B, and let $C = (C_0, C_1, C_2, \ldots, C_t)$ be a sequence of snapshots corresponding to a schedule which adheres to the rules of the red-blue pebble game and satisfies the stopping condition. For each snapshot $C_i \in C$, the following constraint must hold in the WRBPG:*

$$\sum_{v \in \mathcal{R}(C_i)} w_v \leq B$$

All rule-abiding schedules that satisfy the above weighted budget constraint are ***valid schedules*** in the WRBPG. Note that the existence of a valid schedule is not necessarily guaranteed for a given graph, its weights, and budget. We now define the cost of a schedule in terms of its weighted data movements. This cost function is used to find optimum data movement schedules in the WRBPG.

Definition 2.2 (Weighted Schedule Cost). *Let $S^G = (\sigma_1, \ldots, \sigma_t)$ be any valid WRBPG schedule for $G = (V, E, w, B)$. We define $\mathcal{I} = \{\sigma_i \in S^G \mid \sigma_i = M1(v)\}$ to be all $M1$ moves (inputs) in S^G and $\mathcal{O} = \{\sigma_i \in S^G \mid \sigma_i = M2(v)\}$ to be all $M2$ moves (outputs) in S^G. The weighted cost of S^G is defined to be the total sum of node weights for each input/output (I/O) performed during the schedule, i.e.,*

$$Cost(S^G) = \sum_{M1(v) \in \mathcal{I}} w_v + \sum_{M2(v) \in \mathcal{O}} w_v$$

In this cost model, a natural assignment of node weights is the amount of memory needed for the result of the node's operation.

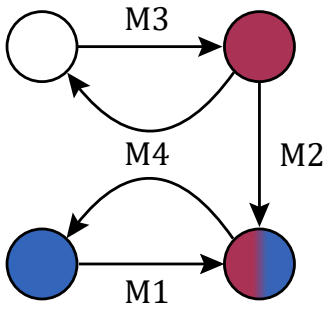


Figure 1: Label transitions for a single node based on the moves of the game. The node labels are *none* (top left), *red* (top right), *blue* (bottom left), and *both* (bottom right).

The weighted schedule cost will then represent the total amount of data transferred between fast and slow memory.

2.2 Basic Properties

In this section, we first show the condition for the existence of a valid WRBPG schedule on a CDAG, and then, we provide a trivial lower bound for any CDAG given the start and stop conditions introduced in Section 2.1.

Proposition 2.3 (Schedule Existence). *Let $G = (V, E, w, B)$ be a node-weighted CDAG with budget B. A valid WRBPG schedule for G exists if and only if, for all $v \in (V \setminus \mathcal{A}(G))$, the following inequality holds:*

$$w_v + \sum_{p \in \mathcal{H}(v)} w_p \leq B$$

Proof. If a valid schedule for G exists, then move $M3(v)$ must have been performed on all non-source nodes $v \in (V \setminus \mathcal{A}(G))$ to satisfy the stopping condition. This move requires the placement of a red pebble on v and its parents $\mathcal{H}(v)$ simultaneously by definition. Thus, $w_v + \sum_{p \in \mathcal{H}(v)} w_p \leq B$ must hold based on the weighted budget constraint in Definition 2.1. Conversely, if $w_v + \sum_{p \in \mathcal{H}(v)} w_p \leq B$ for $v \in (V \setminus \mathcal{A}(G))$, then a schedule can be trivially constructed by computing each node based on a topological ordering and performing moves $M2$ and $M4$ to free up weighted fast memory resources. □

Proposition 2.4 (Algorithmic Lower Bound). *Let $G = (V, E, w, B)$ be any node-weighted CDAG with a valid budget B. For any valid schedule S^G, the $Cost(S^G)$ is lower bounded as follows:*

$$\sum_{v \in \mathcal{A}(G)} w_v + \sum_{v \in \mathcal{Z}(G)} w_v \leq Cost(S^G).$$

Proof. The starting condition of G assumes blue pebbles on all $v \in \mathcal{A}(G)$. Pebbling the graph will therefore require performing move $M1(v)$ on all source nodes at least once, resulting in $\sum_{v \in \mathcal{A}(G)} w_v$ weighted input cost. Similarly, the stopping condition assumes all sink nodes have a blue pebble which must be performed once using move $M2(v)$, resulting in $\sum_{v \in \mathcal{Z}(G)} w_v$ weighted output cost. Since we assume that all source nodes are distinct from sink nodes, the cost of any valid schedule is at least the weighted sum of the sources and sinks. □

2.3 Model Applications

We apply the WRBPG to two key system problems: generating efficient data movement schedules under fast memory constraints, and determining minimal fast memory sizes for application-specific hardware design. Both leverage the weighted schedule cost (Definition 2.2) combined with pebbling algorithms (Section 3) to optimize data transfer between fast and slow memory. Our objective is to find valid schedules that minimize this data movement. We formalize these two optimization targets as follows:

Definition 2.5 (Minimum Weighted Schedule). *Let $\mathcal{S}^G$ be the set of all valid WRBPG schedules on a graph $G = (V, E, w, B)$. A minimal schedule is defined as a schedule $S^G \in \mathcal{S}^G$ that pebbles the graph with the lowest weighted schedule cost, i.e.,*

$$S^G_{\min} = \arg \min_{S^G \in \mathcal{S}^G} Cost(S^G)$$

Definition 2.6 (Minimum Fast Memory Size). *Let $\mathcal{F}$ be the set of all valid budgets on the nodes V, edges E, and node weights w for a given CDAG. We define the minimum fast memory size as the smallest budget $b \in \mathcal{F}$ with a valid WRBPG schedule that matches the algorithmic lower bound (Proposition 2.4), i.e.,*

$$B_{\min} = \min_{b \in \mathcal{F}} b \quad \text{subject to}$$

$$Cost(S^{G^b}_{\min}) = \sum_{v \in \mathcal{A}(G^b)} w_v + \sum_{v \in \mathcal{Z}(G^b)} w_v \quad s.t. \quad G^b = (V, E, w, b)$$

3 Dataflow-Specific Pebbling Algorithms

Optimizing over the space of valid schedules and budgets requires search procedures that are both efficient and agnostic to weighted resources. Procedures which output optimum results for any CDAG in the original red-blue pebble game are known to be PSPACE-complete [14]. The WRBPG simulates the original game when $w_v = 1$ for all $v \in V$ and fast memory budget $B = R$ for any number of red pebbles R, so designing procedures for arbitrary CDAGs to optimally solve the weighted variant is difficult.

Our approach instead focuses on designing dataflow-specific pebbling algorithms since our domain primarily consists of computations with structured graph representations. These structures have regular dataflow patterns which assist our search procedures and allow us to solve the game more efficiently. To highlight our approach, we provide an example graph construction and optimum pebbling algorithm for the DWT kernel. This kernel is an example of a core signal processing workload for BCIs and exhibits a recursive dataflow pattern commonly found across the domain. Our DWT pebbling algorithm exploits the implicit tree structure in the dataflow and efficiently generalizes across weights and all budgets. This result extends to k-ary trees, covering a wide class of dataflows which are the building blocks for most CDAGs, enabling the modeling of multi-operand computations.

3.1 The Discrete Wavelet Transform (DWT)

DWT is a signal processing transformation used for time-frequency analysis on signals and data compression pipelines [24]. The input to the transform is a real-valued signal, and the output of the transform is the scaling function (averages) and the wavelet function (coefficients). In this section, we provide a parameterized dataflow graph for the commonly used Haar wavelet transforms (Definition 3.1). Then, we provide a pebbling algorithm that outputs optimum WRBPG schedules for any instantiation of the dataflow (Algorithm 1).

3.1.1 ***Dataflow*** Given an input vector $\vec{x} \in \mathbb{R}^N$ representing a signal of length N, the Haar wavelet transform will translate $\vec{x}$ into a set of averages $\vec{a}_d \in \mathbb{R}^{N/2^d}$ and a set of coefficients $\vec{c}_d \in \mathbb{R}^{N/2^d}$ for $d = 1, \ldots, \log_2(N)$, where d represents the level of the transform. For $d = 1$, the averages/coefficients are computed using the input vector $\vec{x}$, but for $d > 1$, $\vec{a}_d$ and $\vec{c}_d$ are computed using the previous averages $\vec{a}_{d-1}$ recursively.

Given two consecutive samples $x[j]$ and $x[j+1]$ in the input signal, the average is computed as

$$a[j] = \frac{x[j] + x[j+1]}{\sqrt{2}}$$

and, given the same two samples, the coefficient is computed as

$$c[j] = \frac{x[j] - x[j+1]}{\sqrt{2}}.$$

For level $d = 1$, the first set of averages and coefficients are

$$\vec{a}_1 = \left[\frac{x[0] + x[1]}{\sqrt{2}}, \frac{x[2] + x[3]}{\sqrt{2}}, \ldots, \frac{x[N-2] + x[N-1]}{\sqrt{2}} \right]$$

$$\vec{c}_1 = \left[\frac{x[0] - x[1]}{\sqrt{2}}, \frac{x[2] - x[3]}{\sqrt{2}}, \ldots, \frac{x[N-2] - x[N-1]}{\sqrt{2}} \right]$$

For level $d > 1$, the averages and coefficients are

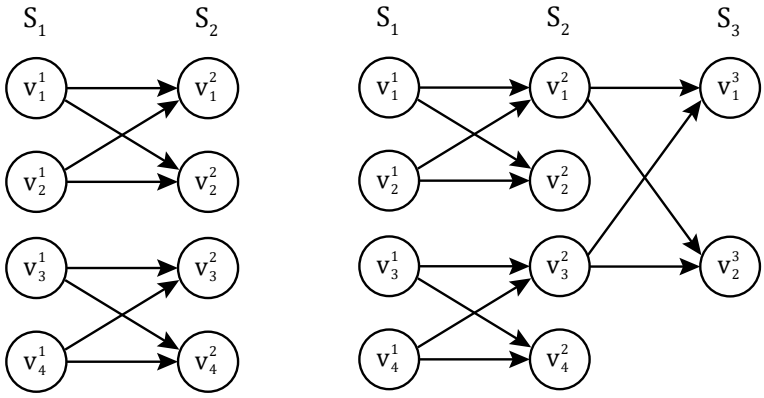
$$\vec{a}_d = \left[\frac{a_{d-1}[0] + a_{d-1}[1]}{\sqrt{2}}, \frac{a_{d-1}[2] + a_{d-1}[3]}{\sqrt{2}}, \ldots, \frac{a_{d-1}[N/2^{d-1} - 2] + a_{d-1}[N/2^{d-1} - 1]}{\sqrt{2}} \right]$$

$$\vec{c}_d = \left[\frac{a_{d-1}[0] - a_{d-1}[1]}{\sqrt{2}}, \frac{a_{d-1}[2] - a_{d-1}[3]}{\sqrt{2}}, \ldots, \frac{a_{d-1}[N/2^{d-1} - 2] - a_{d-1}[N/2^{d-1} - 1]}{\sqrt{2}} \right]$$

This recursion happens for at most $\log_2(N)$ steps, which determines the maximum level for the transform.

Definition 3.1 (DWT Graphs). *Let $DWT(n, d) = (V, E, w, B)$ be the DWT graph consisting of a level $d \in \mathbb{Z}_{\geq 1}$ and inputs $n \in \left\{ k \cdot 2^d \;\middle|\; k \in \mathbb{Z}_{\geq 1} \right\}$. This graph contains $d+1$ sets of nodes $S_1, S_2, \ldots, S_d$, S_{d+1}, where S_1 contains the input nodes and S_{d+1} contains the final set of averages/coefficients. For each S_i where $i > 2$, there are $|S_{i-1}|/2$ nodes in S_i. Suppose we index each S_i from 1 to $|S_i|$. Then, we have the following directed edges:*

1. *For each $v^1_j \in S_1$, where j is the index of the node, there exists a directed edge (v^1_j, v^2_j); also, if $j \bmod 2 = 1$, then there is exists a directed edge $\left(v^1_j, v^2_{j+1}\right)$; otherwise, if $j \bmod 2 = 0$, then there exists a directed edge $\left(v^1_j, v^2_{j-1}\right)$.*
2. *For each $v^i_j \in S_i$ where $2 \leq i \leq d$ and $j \bmod 4 = 1$, there exists directed edges $\left(v^i_j, v^{i+1}_{(j+1)/2}\right)$ and $\left(v^i_j, v^{i+1}_{(j+3)/2}\right)$.*



(a) $DWT(4,1)$ **where** $n = 4$ **and** $d = 1$. **(b)** $DWT(4,2)$ **where** $n = 4$ **and** $d = 2$.

Figure 2: $DWT(n,d)$ **graphs for the same number of inputs** n **but different levels** d **based on Definition 3.1.**

(3) *For each* $v_j^i \in S_i$ *where* $2 \le i \le d$ *and* $j \bmod 4 = 3$, *there exists directed edges* $\left(v_j^i, v_{(j-1)/2}^{i+1}\right)$ *and* $\left(v_j^i, v_{(j+1)/2}^{i+1}\right)$.

Figure 2 shows example graphs based on this definition.

Our DWT construction is for the Haar wavelet and its recursive implementation; however, there are many other wavelets that can be used for the DWT. In general, the DWT performs a convolution with a low pass filter to compute the averages and another convolution with a high pass filter to compute the coefficients. In our example, $\left[\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}\right]$ and $\left[\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]$ are the low-pass and high-pass filters respectively. These convolutions include downsampling and are performed recursively in multiple levels. The dataflow in Definition 3.1 is applicable to any wavelet of size two and any normalization factor (*i.e.*, $\sqrt{2}$, 2, etc.). Furthermore, the non-input nodes in the DWT graph represent addition or subtraction operations plus a scale factor. Coarser or finer operation granularities are possible and functionally equivalent. We opt for finer granularities given our extreme resource constraints. Wavelet transforms that perform convolutions with more than two inputs/averages or coarser operations are left to future work.

3.1.2 ***Optimum Pebbling*** For any given node-weighted DWT(n, d) graph, our proof strategy first prunes the output coefficients of the initial graph such that the pruned graph forms a binary tree. We then introduce a recursive approach to find the minimum schedule for this binary tree. The minimal schedule for the pruned graph is then trivially updated to find the minimal schedule for the initial graph. An example pruning of the initial graph is shown in Figure 3.

Lemma 3.2 (Minimum Weighted Schedule for DWT). *Let* G *be any node-weighted* $DWT(n,d) = (V, E, w, B)$ *graph such that* $w_{v_j^i} \le w_{v_k^i}$ *for* $i > 1$, $j \bmod 2 = 0$, *and* $k \bmod 2 = 1$. *Let* G' *be the graph* G *with all* $v_j^i \in V$ *and* $(u, v_j^i) \in E$ *removed.* $S^G_{\min}$ *can be constructed from* $S^{G'}_{\min}$ *by computing/moving* v_j^i *without excess cost.*

Proof. To compute any node (*i.e.*, move $M3$), there must be red pebbles on their immediate predecessors. Let v be any node $v_j^i \in V$ and u any node $v_{j+1}^i \in V$ where $i > 1$ and $j \bmod 2 = 1$. By construction, v and u have the same immediate predecessors, *i.e.*, $\mathcal{H}(v) = \mathcal{H}(u)$.

Suppose that $\sigma_k = M3(v)$ is the location in the minimum schedule $S^{G'}_{\min} = (\sigma_1, \ldots, \sigma_t)$ where v was computed. Then, we update the schedule by inserting the sequence $(M3(u), M2(u), M4(u))$ after position $k-1$ and before position k. This corresponds to computing u, moving it to slow memory, and then removing it from fast memory. This is possible because the parents of v are shared with u and they are already pebbled red given the valid schedule. Furthermore, we assume $w_u \le w_v$, so the weighted budget constraint remains valid. Inserting these sequences constructs a new schedule S^G_{new} which pebbles all nodes and fulfills the stopping condition since all u nodes are outputs. This implies that $S^G_{new} = S^G_{\min}$ because the only additional costs relative to $S^{G'}_{\min}$ stem from $M2(u)$ which must be performed regardless to fulfill the stopping condition for G, and thus, a schedule with a lower weighted cost does not exist. □

Using this insight, we introduce a procedure for calculating the cost of the minimum weighted schedule for the pruned DWT graph.

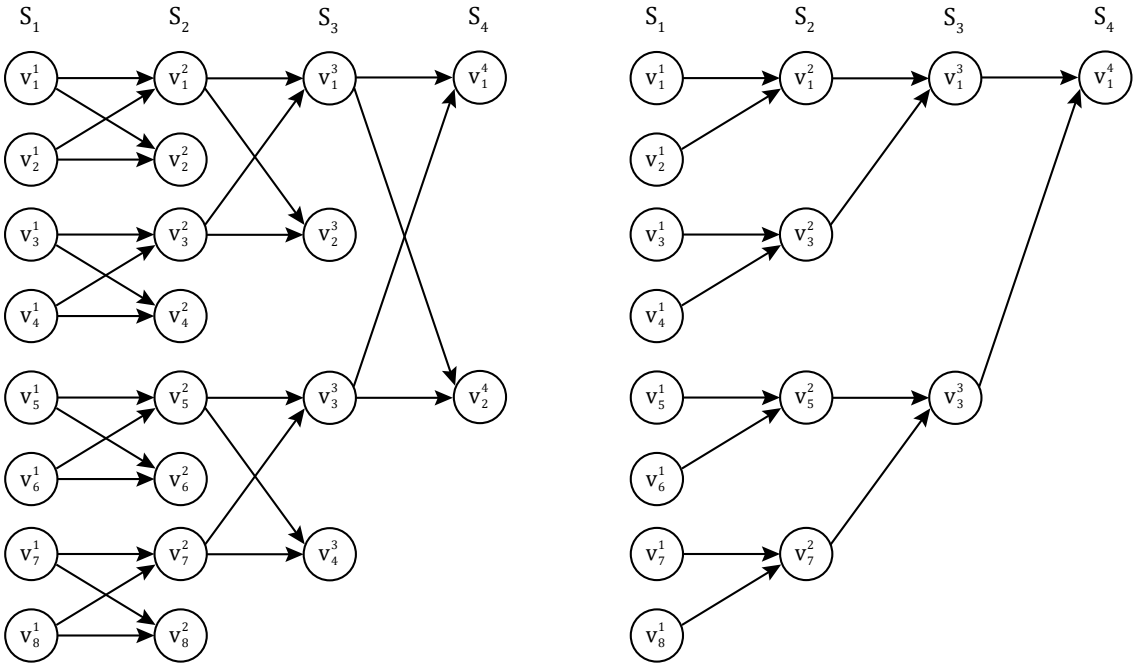
Lemma 3.3 (Minimum Weighted Cost for Pruned DWT). *Let* $S'_1, S'_2, \ldots, S'_d, S'_{d+1}$ *be the layers for any pruned graph* G'. *The cost of the minimum weighted schedule for* G' *with its budget* B *is given by*

$$Cost(S^{G'}_{\min}) = \sum_{v \in S'_{d+1}} w_v + \sum_{v \in S'_{d+1}} P(v, B) \tag{1}$$

where

$$P(v,b) = \begin{cases} \infty & \text{if } w_v + \sum_{p \in \mathcal{H}(v)} w_p > b \\ \min \begin{Bmatrix} P(p_1,b) + P(p_2,b) + 2 \cdot w_{p_1}, \\ P(p_1,b) + P(p_2,b - w_{p_1}), \\ P(p_2,b) + P(p_1,b) + 2 \cdot w_{p_2}, \\ P(p_2,b) + P(p_1,b - w_{p_2}) \end{Bmatrix} & \text{if } p_1, p_2 \in \mathcal{H}(v) \\ w_v & \text{if } \mathcal{H}(v) = \emptyset \end{cases} \tag{2}$$

Proof. The first observation is that DWT graphs could have multiple independent subgraphs based on the values n and d (e.g.,



(a) Original graph G based on Definition 3.1. **(b) Pruned graph G' based on Lemma 3.2.**

Figure 3: The original $DWT(8,3)$ graph G (left) and the pruned graph G' (right) without nodes v_j^i where $j \bmod 2 = 0$ and $i > 1$.

Figure 2a). Placing red pebbles across independent subgraphs concurrently only lowers the budget for each subgraph without improving weighted I/O cost. In other words, any minimum weighted schedule which pebbles subgraphs concurrently can be trivially reordered to have each subgraph pebbled sequentially. Therefore, it is sufficient to produce a minimum weighted schedule for each subgraph given that they all have the same node and edge structure. The cost of the minimum weighted schedule for G' is therefore calculated as the sum of costs across each independent subgraph.

The second observation is that each subgraph has a recursive binary tree structure. We prove the minimum weighted cost of computing this structure using induction. For simplicity, we set the stopping condition to be computing the sink of the (sub)tree by placing a red pebble on it.

Our base case considers the weighted cost of computing layer S_2' in an independent subgraph. Assuming that a valid WRBPG schedule exists, each node $v \in S_2'$ can be computed by moving its parents into fast memory and placing its result in fast memory. Since each $v \in S_2'$ and its parents $p_1, p_2 \in \mathcal{H}(v)$ form an independent subgraph as well, the minimum schedule for each subgraph is to move each parent once using $M1$, which has a cost of $w_{p_1} + w_{p_2}$. This exactly equals $P(v,b) = P(p_1,b) + P(p_2, b - w_{p_1}) = P(p_2,b) + P(p_1, b - w_{p_2}) = w_{p_1} + w_{p_2}$ by definition because $\mathcal{H}(p_1) = \mathcal{H}(p_2) = \emptyset$ and $w_v + w_{p_1} + w_{p_2} \le b$ must hold for any valid budget $b \le B$. If b is not valid, then no WRBPG schedule exists and the cost is set to ∞. Thus, the minimum weighted cost of the entire layer is simply $\sum_{v \in S_2'}(w_{p_1} + w_{p_2}) = \sum_{v \in S_2'} P(v,b)$.

Then, for every node $v^i \in S_i'$ where $i > 2$, we assume via induction that the minimum weighted cost of computing the node v^i is $P(v^i, b)$ and the minimum weighted cost of computing the layer S_i' is $\sum_{v^i \in S_i'} P(v^i, b)$ for any budget $b \le B$. Now, we show the cost of computing the nodes $v^{i+1} \in S_{i+1}'$.

To compute any node v^{i+1}, we must first have a red pebble on the parents $p_1, p_2 \in \mathcal{H}(v^{i+1})$. There are only eight possible strategies for computing v^{i+1}. These are determined by the choice of which parent to compute first in the schedule and whether the parent maintains its red pebble when computing the remaining parent. The best choice will be the lowest weighted cost of all possible strategies. We outline each strategy and cost function as follows:

$$\min \begin{Bmatrix} P(p_1,b) + P(p_2,b) + 2 \cdot w_{p_1} + 2 \cdot w_{p_2}, \\ P(p_1,b) + P(p_2, b - w_{p_1}) + 2 \cdot w_{p_2}, \\ P(p_1,b) + P(p_2,b) + 2 \cdot w_{p_1}, \\ P(p_1,b) + P(p_2, b - w_{p_1}), \\ P(p_2,b) + P(p_1,b) + 2 \cdot w_{p_1} + 2 \cdot w_{p_2}, \\ P(p_2,b) + P(p_1, b - w_{p_2}) + 2 \cdot w_{p_1}, \\ P(p_2,b) + P(p_1,b) + 2 \cdot w_{p_2}, \\ P(p_2,b) + P(p_1, b - w_{p_2}) \end{Bmatrix} \begin{matrix} (1)\ blue\ p_1, blue\ p_2 \\ (2)\ red\ p_1, blue\ p_2 \\ (3)\ blue\ p_1, red\ p_2 \\ (4)\ red\ p_1, red\ p_2 \\ (5)\ blue\ p_2, blue\ p_1 \\ (6)\ red\ p_2, blue\ p_1 \\ (7)\ blue\ p_2, red\ p_1 \\ (8)\ red\ p_2, red\ p_1 \end{matrix} \tag{3}$$

Each entry in this equation corresponds to a sequence of moves. For example, entry (3) corresponds to computing p_1, placing its blue pebble, removing its red pebble, computing parent p_2, then moving p_1 back into fast memory to compute v^{i+1}. We enumerate all possible orderings and whether to keep the red pebble or replace with a blue pebble when computing each parent, ensuring that red pebble placements reduce the budget of the remaining parent.

Furthermore, this equation can be simplified. (3) and (4) will always be a better strategy than (1) and (2) respectively. Similarly, (7) and (8) will always be better than (5) and (6) respectively. Therefore, the representative set of strategies are the following:

$$\min \begin{cases} P(p_1, b) + P(p_2, b) + 2 \cdot w_{p_1}, \\ P(p_1, b) + P(p_2, b - w_{p_1}), \\ P(p_2, b) + P(p_1, b) + 2 \cdot w_{p_2}, \\ P(p_2, b) + P(p_1, b - w_{p_2}) \end{cases} \begin{matrix} (3)\ blue\ p_1, red\ p_2 \\ (4)\ red\ p_1, red\ p_2 \\ (7)\ blue\ p_2, red\ p_1 \\ (8)\ red\ p_2, red\ p_1 \end{matrix} \quad (4)$$

By our induction hypothesis, the minimum weighted cost to compute each parent with a red pebble is $P(p_1, b)$ and $P(p_2, b)$ respectively for any $b \leq B$ since $p_1, p_2 \in S'_i$. These procedures correspond to valid schedules which can be concatenated into the schedule for v^{i+1}. Therefore, the minimum weighted cost of computing v^{i+1} is determined by Eq. (4) which corresponds exactly to $P(v^{i+1}, b)$. This hold for any $v^{i+1} \in S'_{i+1}$ given their independence among each other, and thus, the minimum weighted cost of the layer is $\sum_{v^{i+1} \in S'_{i+1}} P(v^{i+1}, b)$.

Additionally, each output node must be pebbled blue to fulfill the stopping condition of the WRBPG. This can be done as soon as a node $v \in S'_{d+1}$ is computed. The additional cost of this is $\sum_{v \in S'_{d+1}} w_v$ in the schedule. Thus, the cost of the minimum weighted schedule for G' under budget B is $\sum_{v \in S'_{d+1}} w_v + \sum_{v \in S'_{d+1}} P(v, B)$. □

Lemma 3.4 (Minimum Weighted Cost for DWT). *The cost of the minimum weighted schedule for any $DWT(n, d) = (V, E, w, B)$ graph G is calculated as follows:*

$$\sum_{\substack{v_j^i \in V \\ i>1,\ j \bmod 2=0}} w_{v_j^i} + \sum_{\substack{v_k \in S_{d+1} \\ k \bmod 2=1}} w_{v_k} + \sum_{\substack{v_k \in S_{d+1} \\ k \bmod 2=1}} P(v_k, B) \quad (5)$$

Proof. Lemma 3.2 shows that the graph G can be constructed from the minimum schedule of the pruned graph G'. This is done by placing a blue pebble on every pruned node $v_j^i \in V$ where $j \bmod 2 = 0$ and $i > 1$; the total cost of this transformation is $\sum w_{v_j^i}$. The remaining cost of the minimum schedule follows directly from Lemma 3.3. Crucially, this cost is associated with a schedule generation procedure that adheres to the WRBPG rules. □

3.1.3 ***Schedule Generation*** In this section, we bridge the minimum cost characterization established in Lemma 3.4 with a concrete, polynomial-time algorithm that generates an optimal schedule for the DWT. We present an algorithm that produces a minimum-weight schedule while ensuring computational efficiency relative to the input size.

Theorem 3.5 (Optimal Schedule Generation for DWT). *For any instance of the DWT graph $DWT(n, d) = (V, E, w, B)$, Algorithm 1 computes a minimum-weight WRBPG schedule for the graph in time $\Theta(\text{poly}(B \cdot |V|))$.*

Proof. We first establish that Algorithm 1, specifically the function PebbleDWT, produces an optimal schedule for the WRBPG. This result follows directly from Lemma 3.4, which characterizes the construction of a minimum-weight schedule over the DWT graph. The implementation in Algorithm 1 closely mirrors this procedure. In particular, the schedule construction in PebbleTree, defined on Lines 25–38, computes the minimum-weight schedule

Algorithm 1 Optimum WRBPG Schedule Generator for DWT

```
1:  procedure PebbleDWT(G)
2:      M ← ∅                                              ▷ Memo
3:      S^G ← ∅                                            ▷ Schedule
4:      for v_j ∈ Z(G) do
5:          if j mod 2 = 1 then
6:              S^j ← PebbleTree(v_j, B, M) + (M2(v_j), M4(v_j))
7:              S^G ← S^G + S^j
8:          end if
9:      end for
10:     return S^G
11: end procedure
12:
13: procedure PebbleTree(v_j^i, b, M)
14:     v ← v_j^i, u ← v_{j+1}^i
15:     if M[v][b] ≠ ⊥ then
16:         return M[v][b]
17:     end if
18:     if parents(v) = ∅ then
19:         return M1(v)
20:     end if
21:     p_1, p_2 = parents(v)
22:     if w_v + w_{p_1} + w_{p_2} > b then
23:         return INV                                     ▷ Invalid entry, ∞ cost
24:     end if
25:     C ← (M3(u), M2(u), M4(u), M3(v))
26:     P1 ← PebbleTree(p_1, b, M)
27:     P2 ← PebbleTree(p_2, b, M)
28:     LP1 ← PebbleTree(p_1, b − w_{p_2}, M)
29:     LP2 ← PebbleTree(p_2, b − w_{p_1}, M)
30:     R1 ← M1(p_1), R2 ← M1(p_2)
31:     B1 ← M2(p_1), B2 ← M2(p_2)
32:     M[v][b] ← ArgMin(                                  ▷ Returns schedule
33:         Cost(P1 + B1 + P2 + R1 + C),
34:         Cost(P1 + LP2 + C),
35:         Cost(P2 + B2 + P1 + R2 + C),
36:         Cost(P2 + LP1 + C)
37:     )
38:     return M[v][b]
39: end procedure
```

for each subtree rooted at a node. Line 25 handles the moves required for the pruned node and includes the computation of the current node. The base case of the recursion is implemented on Lines 18–20, while the budget feasibility check is enforced on Lines 22-24. Memoization of subproblem solutions is performed in Lines 15–17. Finally, the outer PebbleDWT function iterates over the output nodes in Lines 4–9. Line 6 performs the final blue pebble placement for the outputs of each tree.

We now analyze the runtime of Algorithm 1. The PebbleDWT procedure iterates over every other output node in the graph, pebbling each independent subgraph of the DWT. Each recursive call requires at most $B \cdot |V|$ operations due to the memoization. The algorithm stores the minimum-weight schedule for every node at

$$P_t(v,b) = \begin{cases} \infty & \text{if } w_v + \sum_{p\in\mathcal{H}(v)} w_p > b \\ \min\limits_{\substack{\delta\in\{0,1\}^k \\ \sigma\in\mathrm{Perm}(\mathcal{H}(v))}} \left\{ \sum\limits_{i=1}^{k} P_t(\sigma(i), b - \sum\limits_{j=1}^{i-1} \delta_j \cdot w_{\sigma(j)}) + 2 \cdot \sum\limits_{i=1}^{k} (1-\delta_i)\cdot w_{\sigma(i)} \right\} & \text{if } \mathcal{H}(v) \neq \emptyset \\ w_v & \text{if } \mathcal{H}(v) = \emptyset \end{cases} \tag{6}$$

each possible budget value, avoiding redundant computations. Constructing individual moves within the schedule occurs in constant time, while evaluating the cost of a schedule naively takes time linear in the number of nodes; however, this cost can be stored and updated incrementally with each step to maintain constant-time access. Since the total number of iterations is bounded by $O(B\cdot|V|)$ and each operation within an iteration executes in polynomial time, the overall runtime of the algorithm lies within $\Theta(\mathrm{poly}(B\cdot|V|))$. □

This completes our dataflow-specific algorithm design for the DWT. Through this example, we demonstrate how the WRBPG framework can be effectively applied to generate schedules that are both provably optimal and computationally efficient for relevant classes of graphs. We further leverage this procedure in our evaluation to compare I/O costs (Section 5.2) and to analyze minimum fast memory requirements (Section 5.3).

3.2 k-ary Trees

Our recursive approach to optimal schedule generation is broadly applicable to the class of k-ary tree graphs. In this section, we extend our recursive cost procedure for the minimum weighted schedule to handle arbitrary in-degrees instead of considering only binary trees. This approach is efficient up to some bound on the in-degree relative to the inputs.

Definition 3.6 (k-ary Tree Graphs). *Let $k \in \mathbb{N}_{\geq 1}$. A* k-ary tree graph *is a node-weighted graph $T = (V, E, w, B)$ under the resource budget B with the following properties:*

1. *T is a rooted tree with a unique* sink node *$r \in V$ such that every node in $V \setminus \{r\}$ has a directed path to r.*
2. *Each node $v \in V$ has at most k incoming edges:*
$$\text{in-deg}(v) \leq k.$$
3. *Each edge $(u, v) \in E$ is directed from a* parent *node u to its* child *v, forming a computation directed toward the sink r.*

We denote the class of such graphs with parameter k as $\mathcal{T}_k$.

Lemma 3.7 (Minimum Weighted Cost for k-ary Trees). *Let $T = (V, E, w, B) \in \mathcal{T}_k$ be a weighted k-ary tree graph with n nodes, where each node has in-degree at most k. The cost of the minimum weighted schedule of T for its sink node $r \in V$ and its budget B is given as follows:*

$$Cost(S^T_{\min}) = w_r + P_t(r, B) \tag{7}$$

where $P_t(v, b)$ is defined in Equation (6).

Proof. We prove this calculation through induction similar to Lemma 3.3. For simplicity, we set the stopping condition to be computing the sink of the (sub)tree by placing a red pebble on it. The base case considers any input node $v \in \mathcal{A}(T)$. Placing a red pebble on an input node requires moving them into fast memory using $M1(v)$ with a trivial weighted cost of w_v.

Assume via induction that the minimum weighted cost of the computing any subtree rooted at node v^h with h levels is $P_t(v^h, b)$ for any budget $b \leq B$. The cost of computing the subtrees rooted nodes with $h+1$ levels is as follows. Each root will have up to k parents, *i.e.*, $p_1, p_2, \ldots, p_k \in \mathcal{H}(v^{h+1})$, and each parent will have h levels by definition. The minimum cost strategy will be to enumerate ordering of the parents, and for each ordering, selecting whether to keep the red pebble on a parent or placing a blue pebble on it instead. This is a binary decision so there are 2^k possible placement choices. Combined with the $k!$ possible orderings, there are $2^k(k!)$ possible schedules. The best schedule is the lowest weighted cost among them. By our induction hypothesis, $P_t(p_i, b)$ for $1 \leq i \leq k$ is the minimum cost schedule for each parent, and therefore, $P_t(v^{h+1}, b)$ calculates the minimum schedule cost for any subtree with $h+1$ levels. The final step is to place a blue pebble on the root $r \in V$ of the k-ary tree T. This cost is simply w_r. Therefore, *Eq.* (7) is the optimum weighted cost for any k-ary tree $T \in \mathcal{T}_k$. □

We now show that this optimal procedure is efficient up to some bound on k. Together, this translates directly to polynomial-time schedule generation for most computational k-ary trees with any resource budget and assignment of node weights.

Theorem 3.8 (Optimal Schedule Generation for Bounded In-degree Trees). *Let $T = (V, E, w, B) \in \mathcal{T}_k$ be a weighted k-ary tree graph with n nodes, where each node has in-degree at most k. Then, for any budget B, the optimal schedule for T under the WRBPG can be constructed in polynomial time, provided that $k = O(\log\log n)$.*

Proof. The optimality proof in Lemma 3.7 performs up to $2^k(k!)$ recursive steps for each node in the tree. If k is $O(\log\log n)$, then this is $O(\log n \cdot (\log\log n)!)$ steps for each node. By Stirling's Approximation, $(\log\log n)!$ is bounded as follows (assuming base-2 logarithm):

$$2^{\log((\log\log n)!)} = 2^{O(\log\log n \cdot \log\log\log n)}$$

which is further bounded by $O(n)$, *i.e.*,

$$\begin{aligned} 2^{O(\log\log n\cdot\log\log\log n)} &= 2^{O((\log\log n)^2)} \\ &= 2^{O(2^{\log\log n})} \\ &= 2^{O(\log n)} \\ &= O(n) \end{aligned}$$

Thus, $(\log\log n)! = O(n)$, and the number of recursive calls is $O(n\log n)$ for each node at a given budget. Through dynamic programming, the minimum schedule cost can be cached for each node and budget. The total runtime is therefore $O(B\cdot|V|^2\log|V|)$ assuming $k = O(\log\log n)$. In practice, k is typically $O(1)$ with

$$P_m(v,b,I,R) = \begin{cases} \infty & \text{if } \sum_{r\in R\cup\mathcal{H}(v)\cup\{v\}} w_r > b \\ \min\begin{cases} P_m(p_1, b-\sum_{j\in I_{p_2}} w_j, I_{p_1}, R_{p_1}) + P_m(p_2, b-\sum_{r\in R_{p_1}} w_r, I_{p_2}, R_{p_2}) + 2\cdot w_{p_1}, \\ P_m(p_1, b-\sum_{j\in I_{p_2}} w_j, I_{p_1}, R_{p_1}) + P_m(p_2, b-\sum_{r\in R_{p_1}\cup\{p_1\}} w_r, I_{p_2}, R_{p_2}), \\ P_m(p_2, b-\sum_{j\in I_{p_1}} w_j, I_{p_2}, R_{p_2}) + P_m(p_1, b-\sum_{r\in R_{p_2}} w_r, I_{p_1}, R_{p_1}) + 2\cdot w_{p_2}, \\ P_m(p_2, b-\sum_{j\in I_{p_1}} w_j, I_{p_2}, R_{p_2}) + P_m(p_1, b-\sum_{r\in R_{p_2}\cup\{p_2\}} w_r, I_{p_1}, R_{p_1}) \end{cases} & \text{if } v\notin I \text{ and } p_1,p_2\in\mathcal{H}(v) \\ w_v & \text{if } v\notin I \text{ and } \mathcal{H}(v)=\emptyset \\ \sum_{r\in(R\setminus I)} w_r & \text{if } v\in I \end{cases} \tag{8}$$

respect to n and many recursive calls are redundant during the permutations. □

4 Fast Memory States and Data Reuse

Many computational dataflows have nodes with out-degree more than one. Finding the minimum cost schedule for these CDAGs requires exploring data reuse. Data is said to be reused when a value is kept in fast memory across consecutive operations—avoiding I/Os which are expensive in time and energy. Exploiting data reuse must consider multiple options regarding the choice of operations, their order, when to keep a value in fast memory, and when to release it.

To incorporate data reuse, we extend our k-ary tree pebbling algorithm to schedule under arbitrary fast memory states before and after computing a node. We then apply these procedures to the Matrix-Vector Multiplication (MVM) graph defined in Section 4.2. This kernel represents a fundamental operation to compare/classify independent signals with opportunities for data reuse. We tile MVM using our memory state extensions to optimize data reuse. Our data reuse approach not only extends to dense and structured sparse tensor multiplication, but to less regular CDAGs as well.

4.1 Initial and Reuse States

In this section, we augment the recursive pebbling procedure to include user-defined memory states. The user provides an initial state, a reuse state, a budget, and a node v to compute. Initial states are subsets of nodes which already exist in fast memory before computing v. Reuse states are subsets of nodes who should be present in fast memory after v has been computed. The k-ary scheduling procedure can be extended to include these states into its input conditions. For simplicity, we will take the case where $k = 2$.

Let $T = (V, E, w, B) \in \mathcal{T}_k$ be a weighted k-ary tree graph with root $r \in V$. Let $X \subseteq V$ be any subset of nodes and $u \in V$ be any in the graph, then Let X_u denote the subset of nodes in X that are either predecessors of a node u or u itself in the tree T, *i.e.*,

$$X_u \triangleq X \cap (\text{pred}_T(u) \cup \{u\})$$

We set $I \subseteq V$ to be an initial memory state and $R \subseteq V$ be a reuse memory state provided by the user. The procedure $P_m(r, B, I_r, R_r)$ returns the cost of the WRBPG schedule under these memory state semantics. Eq. (8) includes changes to the minimum cost procedure for scheduling the pruned DWT graph in Eq. (2). These changes are explained as follows.

From the perspective of the current node being computed v, there are two cases: whether the node has already been computed in the initial memory state or not. If $v \in I$, then we do not need to move v, but we need to ensure that all nodes that are in the reuse state R are present in fast memory. We assume that these nodes have blue pebbles on them and do not need to be recomputed. $\sum_{r\in(R\setminus I)} w_r$ brings these nodes into fast memory to preserve the reuse semantics while considering the initial state of memory.

If $v \notin I$, then we consider whether v is an input node or whether it has parents. If v is an input node, then we need to bring it into fast memory which incurs a cost of w_v. If v is not an input node, then we need to incorporate these memory states into the budget of the individual parents $p_1, p_2 \in \mathcal{H}(v)$ based on their ordering. The parent that is computed *first* must take into account the weighted cost of all nodes that are in the initial memory state from the remaining parent. Similarly, the parent that is computed *second* must take into account the weighted cost of all nodes that are reused and kept in fast memory from the first parent. Given that each parent and its predecessors forms an independent subgraph and are computed sequentially, these budget adjustments occur recursively and each parent need only consider its predecessor nodes for its initial and reuse states.

Finally, the budget constraint must include the current node v, its parents $\mathcal{H}(v)$, and the nodes in its reuse set R_v for the weighted red pebble constraint. This is because all of these nodes must at some point be present in fast memory to preserve the memory state semantics. We additionally make the assumption that once a node $r \in R$ is computed or brought into fast memory, it remains in fast memory. Other valid schedules exist which can perform additional I/Os for the reuse nodes after v and its parents have been computed; however, this is typically not common when trying to reuse an already computed result.

We use Eq. (8) and its derivatives to model data reuse in our pebbling algorithms. By providing sets of initial and reuse nodes (*e.g.*, accumulator results, nodes with out-degree greater than one, *etc.*), we can optimize over different WRBPG schedules to find those that produce the least weighted cost. We use these insights to implement a tiling approach for Matrix-Vector Multiplication in Section 4.3.

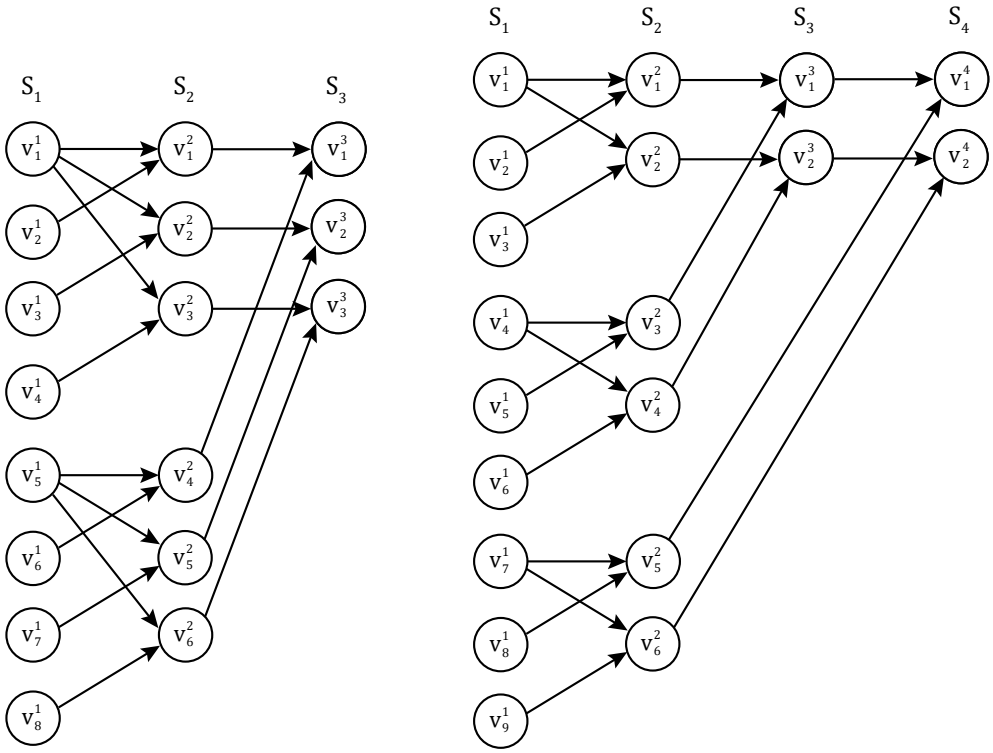
4.2 Matrix-Vector Multiplication (MVM)

MVM computes the product of a matrix $A \in \mathbb{R}^{m\times n}$ and a vector $\vec{x} \in \mathbb{R}^n$, producing an output vector $\vec{y} \in \mathbb{R}^m$ such that

$$y_i = \sum_{k=1}^{n} a_{ik}x_k, \quad \text{for } i = 1,\ldots,m.$$

Based on this definition, we construct an equivalent CDAG representation for arbitrary matrix dimensions m and n.

Definition 4.1 (MVM Graphs). *Let $MVM(m,n) = (V, E, w, B)$ be the MVM graph consisting of a number of rows $m \in \mathbb{Z}_{\geq 2}$ and a*



(a) $MVM(3,2)$ **where** $m = 3$ **and** $n = 2$. **(b)** $MVM(2,3)$ **where** $m = 2$ **and** $n = 3$.

Figure 4: $MVM(m,n)$ **graphs with** m **rows and** n **columns based on Definition 4.1.**

number of columns $n \in \mathbb{Z}_{\geq 1}$. *This graph contains* $n+1$ *sets of nodes* $S_1, S_2, \ldots, S_n, S_{n+1}$, *where* S_1 *are the input nodes and* S_{n+1} *are the output nodes. For* S_1, *there are* $mn + n$ *nodes corresponding to all matrix and vector inputs. For* S_2, *there are* mn *nodes for the initial products of matrix-vector entries, and for* S_i *where* $i > 2$, $|S_i| = m$ *for the accumulation operations. Suppose we index the sets of nodes in each* S_i *from* 1 *to* $|S_i|$. *Then, we have the following directed edges:*

(1) *For each* $v_j^1 \in S_1$ *and* $k = \lfloor (j-1)/(m+1) \rfloor$, *if* $j \bmod (m+1) = 1$, *there is a directed edge* (v_j^1, v_{j-k+i}^2) *for* $i = 0, 1, \ldots, m-1$; *if* $j \bmod (m+1) \neq 1$, *there is a directed edge* (v_j^1, v_{j-k-1}^2).
(2) *For each* v_j^i *where* $2 \leq i \leq n$ *and* $1 \leq j \leq m$, *there is a directed edge* (v_j^i, v_j^{i+1}).
(3) *For each* v_j^2 *where* $m < j \leq mn$, *if* $j \bmod m = 0$, *there is a directed edge* $(v_j^2, v_m^{2+\lfloor (j-1)/m \rfloor})$; *if* $j \bmod m \neq 0$, *there is a directed edge* $(v_j^2, v_{j \bmod m}^{2+\lfloor (j-1)/m \rfloor})$.

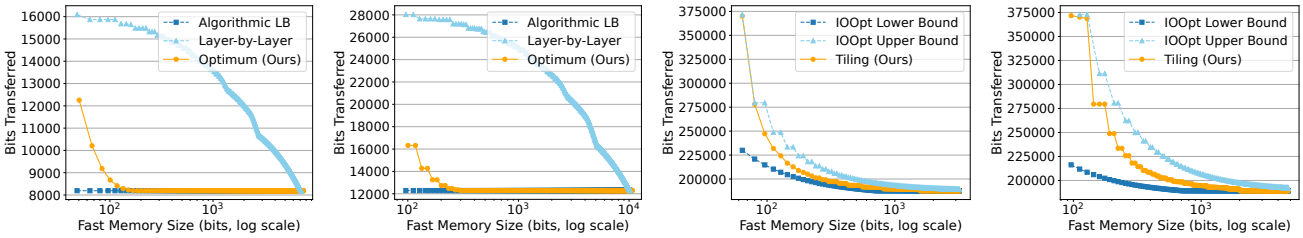
Some examples of this construction are shown in Figure 4.

4.3 Dataflow-Specific Tiling

In this section, we provide an overview for a tiling approach designed using initial/reuse memory states. This forms the basis for our evaluation in Section 5. For $MVM(m,n)$, this dataflow-specific pebbling algorithm constructs minimal WRBPG schedules for sections of the graph called tiles. The schedules for these tiles are then stitched together in a particular order to build the schedule for the entire graph. In this case, we consider two-dimensional tiles with a tile height and a tile width. The tile height in the context of the $MVM(m,n)$ graph construction corresponds to the number of output trees being pebbled concurrently. The tile width corresponds to the relative depth of the output trees to compute by the end of the tile schedule.

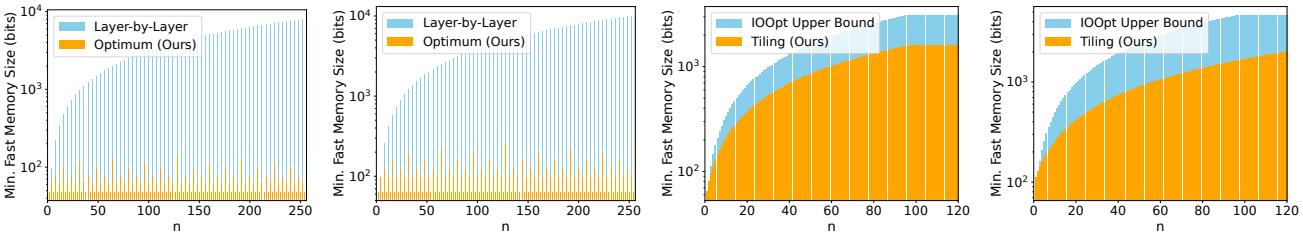
For each tile, our algorithm uses the k-ary tree procedure (for $k = 2$) with initial/reuse memory states shown in Eq. (8). The overall approach is parameterized for any tile height and width and accepts arbitrary memory states. The tile sizes to optimize over are based on the memory budget and workload dimensions m and n. Different combinations of memory reuse, both in terms of the m outputs and n vector inputs in the matrix-vector product, are considered, and the minimal weighted schedule is selected among them.

The tiling strategy that performs the best in most cases is when the width has size one and the height is a function of the budget B. In our case, the tile height represents the number of accumulators simultaneously in fast memory. If m accumulators can fit into fast memory, plus extra space to perform the computations based on the rules of the pebble game, then this corresponds to the lowest weighted cost when $m < n$. If $n < m$, then giving priority to the vector in fast memory will lead to a smaller fast memory size. With arbitrary node weights, the relative cost of an accumulator must be taken into account when deciding how many accumulators should be placed in fast memory versus the vector. For smaller fast memory



(a) ***Equal*** $DWT(256, 8)$ **(b)** ***DA*** **DWT(256, 8)** **(c)** ***Equal*** $MVM(96, 120)$ **(d)** ***DA*** $MVM(96, 120)$

Figure 5: Bits transferred between fast and slow memory as a function of fast memory size. The bitwidth for each input node and memory location (*i.e.*, word) is set to 16 bits, a common sample size for BCI sensor data. *DA* refers to *Double Accumulator*.



(a) ***Equal*** $DWT(n, d^*)$ **(b)** ***DA*** $DWT(n, d^*)$ **(c)** ***Equal*** $MVM(96, n)$ **(d)** ***DA*** $MVM(96, n)$

Figure 6: Minimum fast memory size (log scale), which is when the I/Os are equal to the algorithmic lower bound, as a function of the workload parameter n. The memory word size is 16 bits. For $DWT(n, d^*)$, d^* is set to the largest level possible for the value of n. *DA* refers to *Double Accumulator*.

sizes, there are trade-offs between tile height and vector reuse. For example, if the largest tile height for accumulators is less than m, then it may be better to reduce the tile height and give the extra fast memory space to the vector.

These choices are included in the tiling scheduler by searching over different ways of computing a single tile, orderings across tiles, tile sizes, and memory states that could be preserved after computing a tile, including boundary conditions. Furthermore, because the underlying pebbling algorithms are based on dynamic programming, there is significant overlap among subproblems across these choices. Overall, this tiling approach enables fine-grained data movement optimization over the dataflow and is extensible to more complicated tensor computations and their graph representations.

5 Evaluation

We evaluate the weighted schedules derived from our WRBPG model, focusing on weighted I/O costs — accounting for data transfers with varying precision — and the minimum fast memory size defined in Definition 2.6. For these fast memory sizes, we perform physical memory synthesis to generate circuit-level designs from our theoretical models, enabling measurement of power consumption (static and dynamic), performance, and silicon area.

5.1 Experimental Setup

We evaluate our scheduling approach using two benchmark graphs: $DWT(256, 8)$ and $MVM(96, 120)$, as defined in Definition 3.1 and Definition 4.1, respectively. For the DWT graph, we set $n = 256$ and $d = 8$, where d represents the maximum number of wavelet decomposition levels for an input of size 256. For the MVM graph, we set $m = 96$ and $n = 120$, corresponding to a typical configuration in BCI systems, where 96 electrodes from a Utah array are used to process neural signals at 20–30 kHz sampling rates within millisecond latency constraints.

We evaluate each benchmark graph under two node weight configurations: *Equal* and *Double Accumulator*. In the *Equal* configuration, all nodes are assigned the same weight, corresponding to the classic unweighted red-blue pebble game [22]. In the *Double Accumulator* configuration, each non-input node—representing a partial or accumulated result—has twice the weight of an input node. This reflects a common mixed-precision scenario where accumulated values require higher numerical precision than raw inputs. In both configurations, input nodes are assigned 16-bit weights. In the *Double Accumulator* configuration, accumulated values are assigned 32-bit weights.

For $MVM(96, 120)$, we compare our optimal tiling-based approach (Section 4.3) against IOOpt, a memory-constrained I/O minimization strategy from prior work [34, 35]. For $DWT(256, 8)$, we compare against a layer-by-layer baseline instead of IOOpt, because it does not support the recursive dataflow structure of DWT.

The layer-by-layer baseline schedules nodes sequentially by traversing graph layers S_2 through S_{d+1}. Within each layer S_k, nodes are scheduled in index order from 1 to $|S_k|$. When the fast memory budget is exceeded, nodes that are red-pebbled but not yet

Table 1: Minimum fast memory size comparison among the workloads in Fig. 5 (* indicates our proposed approaches).

Workload	Node Weights	Scheduling Approach	Minimum Fast Memory Size (words)	Word Size (bits)	Minimum Capacity (bits)	Power-of-Two Capacity (bits)
DWT(256, 8)	Equal	Optimum*	10	16	160	256
DWT(256, 8)	Equal	Layer-by-Layer	445	16	7120	8192
DWT(256, 8)	Double Accumulator	Optimum*	18	16	288	512
DWT(256, 8)	Double Accumulator	Layer-by-Layer	636	16	10176	16384
MVM(96, 120)	Equal	Tiling*	99	16	1584	2048
MVM(96, 120)	Equal	IOOpt UB	193	16	3088	4096
MVM(96, 120)	Double Accumulator	Tiling*	126	16	2016	2048
MVM(96, 120)	Double Accumulator	IOOpt UB	289	16	4624	8192

used by their children are spilled to slow memory in first-in, first-out (FIFO) order based on when they were placed in fast memory. If a node has no remaining children to compute, its red pebble is deleted or, if it is an output node, moved to slow memory. To reduce I/O costs, we alternate the traversal direction in each layer: ascending index order $(1, \ldots, |S_k|)$ followed by descending $(|S_{k+1}|, \ldots, 1)$ in the next layer. This optimization helps retain recently computed values across adjacent layers.

All evaluated scheduling strategies—optimal, layer-by-layer, and tiling—are implemented in C++ and designed to construct valid WRBPG schedules under any fast memory constraint.

5.2 I/O Comparison

In this section, we evaluate the weighted I/O cost of each workload described in Section 5.1 as a function of the fast memory size. Figure 5 reports the number of bits transferred for each workload under varying memory budgets. Across all configurations and memory sizes, our algorithms consistently achieve lower I/O cost compared to prior methods. We also include two lower bounds for reference: the *Algorithmic Lower Bound* and the *IOOpt Lower Bound* [35] for $DWT(256, 8)$ and $MVM(96, 120)$, respectively.

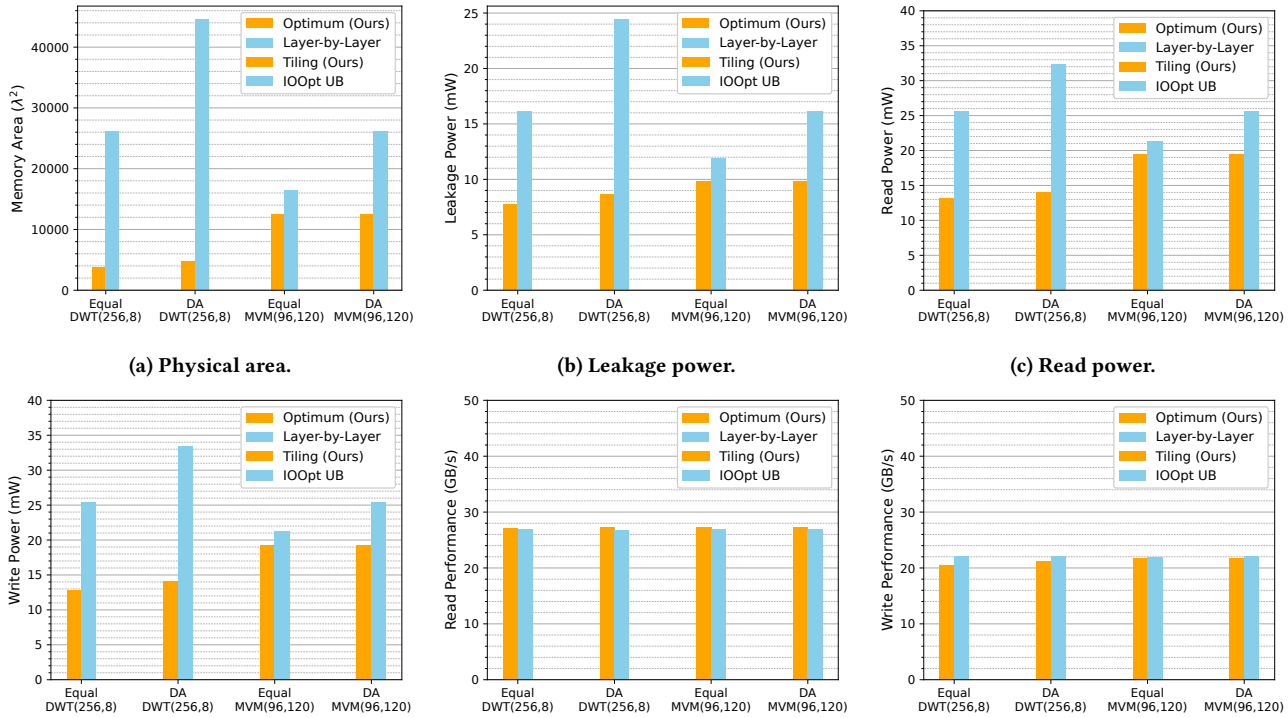
The Algorithmic Lower Bound, defined in Proposition 2.4, is the weighted sum of all inputs and outputs—*i.e.*, all source and sink nodes of the CDAG. This bound is widely used in hardware and systems design as a best-case estimate of I/O complexity [20]. The IOOpt Lower Bound, in contrast, is automatically derived using a geometric and parametric analysis of loop nests under the polyhedral model.

IOOpt, however, presents two limitations in our analysis. First, it is not directly applicable to recursive graph structures such as $DWT(256, 8)$, since it assumes loop nest representations. This limitation also prevents us from using IOOpt's upper bound for comparison, motivating our use of the layer-by-layer scheduling baseline instead. Second, IOOpt does not support weighted or mixed-precision schedules, making it unsuitable for evaluating the *Double Accumulator* variant of $MVM(96, 120)$. To compensate, we manually adjust IOOpt's bounds in the following way: for the lower bound, we double the weight of each accumulator output (*i.e.*, multiply the output term by 2); for the upper bound, we assume all non-input/output data movements are double-weighted. To remain conservative, we avoid making assumptions about the exact proportion of single- vs. double-weighted transfers. Additionally, we increase the memory budget in IOOpt's model to include twice the accumulator allocation used in their original fast memory split (which typically gives just under half to outputs).

Both of our methods—*optimum* and *tiling*—outperform or match the respective baselines (*layer-by-layer* and *IOOpt* [34]) across all fast memory sizes. The superiority of our optimum algorithm over the layer-by-layer baseline is guaranteed by Theorem 3.5, which proves that the schedule is both a lower and upper bound—hence optimal—for all weight assignments on $DWT(256, 8)$. The tiling approach used for $MVM(96, 120)$ surpasses IOOpt for two reasons. First, IOOpt allocates a fixed ratio of fast memory to inputs and outputs, typically splitting memory in half, while our tiling method flexibly assigns fast memory based on tile height. This allows us to prioritize outputs as long as sufficient input reuse is maintained, enabling more rows to be processed with fewer I/Os. Second, IOOpt requires each of the m output values to be read and written, whereas our tiling approach writes each output exactly once—eliminating a significant portion of unnecessary transfers.

5.3 Minimum Fast Memory Size Comparison

We now evaluate the minimum fast memory size required for each scheduling approach. Specifically, we report the smallest fast memory capacity at which the schedule achieves the I/O lower bound from Figure 5. These results are summarized in Table 1 for the workloads defined in Section 5.1.



(a) Physical area. **(b) Leakage power.** **(c) Read power.**

(d) Write power. **(e) Peak read performance.** **(f) Peak write performance.**

Figure 7: Memory read power, write power, leakage power, read performance, write performance, and physical area based on the power-of-two capacities in Table 1, synthesized with AMC in TSMC 65 [2]. *DA* refers to *Double Accumulator*.

We conduct two additional analyses. First, Figure 6 shows how the minimum fast memory size scales with the problem size parameter n, which reflects the number of nodes in the underlying tree of each graph. Second, we study how the reduced memory requirements translate into concrete hardware benefits—power, performance, and area—through physical synthesis of SRAM designs. These results appear in Figure 7, with corresponding transistor-level layouts in Figure 8. Across all configurations, our approaches consistently yield smaller fast memory requirements than prior methods.

As shown in Table 1, for $DWT(256, 8)$, our optimum scheduling algorithm reduces the required memory size by 97.8% and 97.2% compared to the layer-by-layer baseline under the *Equal* and *Double Accumulator* weightings, respectively. For $MVM(96, 120)$, our tiling algorithm achieves a memory size reduction of 48.7% and 56.4% compared to the IOOpt Upper Bound [34].

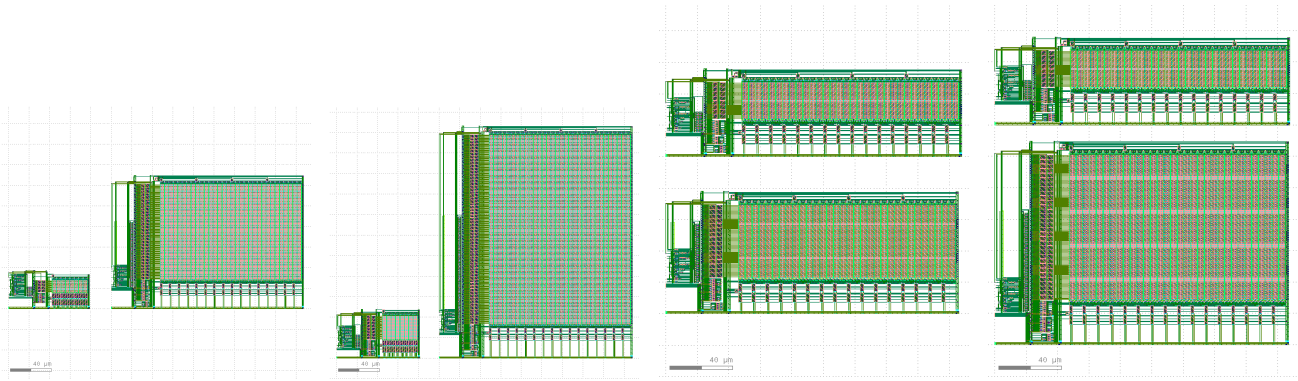
To examine scaling behavior, we vary the problem size. For $DWT(n, d)$, we set $2 \leq n \leq 256$ with even values of n, and select the maximum resolution level d as the largest power of two less that is a multiple of n. On average, our optimum approach reduces the minimum memory size by 47.3% and 46.8% under the *Equal* and *Double Accumulator* settings, respectively. For $MVM(m, n)$, we fix $m = 96$ and vary $1 \leq n \leq 120$. Our tiling method yields average memory size reductions of 18.6% and 36.2% respectively.

To assess how our reduced memory sizes impact hardware implementation, we synthesize SRAM arrays using the TSMC 65 nm process node. We employ a variant of AMC [2], an open-source memory compiler specialized for this process. The synthesis results in Figure 7 reveal reductions in area, leakage power, read power, and write power, while maintaining comparable performance.

For synthesis, we round each required memory size to the nearest power of two (a standard design practice). These rounded values appear in the final column of Table 1. Figure 7a shows the physical area in λ^2, a standard unit in layout scaling [46]. For $DWT(256, 8)$, our designs reduce physical area by 85.7% and 89.5% for the *Equal* and *Double Accumulator* variants, respectively—a 32× difference in power-of-two size. Layouts are shown in Figures 8a and 8b.

For $MVM(96, 120)$, area reductions are 24.3% and 52.6% for *Equal* and *Double Accumulator* respectively, corresponding to 2× and 4× power-of-two differences. These are illustrated in Figures 8c and 8d. The average area reduction is 63% across our workloads. Notably, our tiling approach equalizes memory capacity across both variants, unlike IOOpt, which incurs a 2× overhead in the *Double Accumulator* case. This demonstrates our scheduler's ability to better utilize memory under mixed-precision settings.

Figure 7b presents the leakage power comparison, showing an average reduction of 43.4% mW across workloads, with a peak savings of 15.8 mW. Read and write power reductions are shown in Figures 7c and 7d, averaging 34.6% and 35.4% mW, respectively,



(a) ***Equal*** **DWT(256, 8). Optimum (left, 256 bits) versus layer-by-layer (right, 8192 bits).**

(b) ***DA*** **DWT(256, 8). Optimum (left, 512 bits) versus layer-by-layer (right, 16384 bits).**

(c) ***Equal*** **MVM(96, 120). Our tiling approach (top, 2048 bits) versus IOOpt (bottom, 4096 bits).**

(d) ***DA*** **MVM(96, 120). Our tiling approach (top, 2048 bits) versus IOOpt (bottom, 8192 bits).**

Figure 8: Physical layout comparison between power-of-two memory sizes among the different approaches described in Section 5.1 using the results in Table 1. ***DA*** **refers to** ***Double Accumulator.***

with peaks at 18.3 and 19.3 mW. Read/write throughput remains nearly constant (Figures 7e and 7f) due to fixed synthesis parameters and gate sizing in AMC. Though these settings can be further optimized, they are sufficient to highlight the primary benefit: our Weighted Red-Blue Pebble Game-based schedulers significantly reduce memory area and power usage without performance degradation—crucial for embedded, power-constrained hardware designs.

6 Related Work

Pebble games are combinatorial abstractions which have a rich history of being used to model various problems in computing. First introduced to model register allocation [13, 39] and storage costs [40], they have since been used to study I/O complexity [14, 22, 27, 28, 31], high performance computation [17, 29, 42], reversible computation [3, 38], non-deterministic straight-line programs [11, 19], proof complexity [8, 9, 12, 32] and, more recently, re-materialization costs in deep neural networks [21, 26], peak memory scheduling [23], and cryptographic applications [4–6, 16]. Despite their expressive power, finding an optimal schedule in the red-blue pebble game is known to be PSPACE-hard [14], and many natural variants are also computationally intractable [14, 36].

Jin et al. [23] recently introduced a weighted variant of the standard (non-red-blue) pebble game, providing optimal algorithms for specific graph families like series-parallel graphs, and new hardness results showing that optimal schedules are difficult to compute on pumpkin graphs. Their work also introduced the notion of *dominating schedules*, which guarantee lower memory usage than all alternatives. While they focus on the standard pebble game, our work extends the red-blue pebble game to the weighted setting.

To our knowledge, Chen et. al. [10] is the only work that has applied red-blue pebble games directly to hardware design. Their focus is on deriving asymptotic I/O lower bounds for convolutional operators using loop nest representations, which inform hardware mappings and memory hierarchy designs. In contrast, our work emphasizes deriving provably optimal schedules—upper bounds—for classes of CDAGs in a weighted model. While lower bounds provide valuable theoretical limits, they may significantly underestimate real I/O behavior in power- and memory-constrained settings. Our results show that in such regimes, the gap between lower bounds and feasible schedules can widen, especially when mixed-precision or data-dependent weights are introduced.

7 Conclusion

This paper introduces the Weighted Red-Blue Pebble Game, a novel approach for the co-design of algorithms and architectures in highly resource-constrained systems. Using representative kernels from the domain of implantable BCIs—a setting with extreme power and energy constraints—we develop efficient scheduling strategies for modular computational constructs within the CDAGs of these workloads. In particular, we derive exact, optimum schedules for a broad class of tree-based CDAGs, which can be composed to scale across diverse applications. Through detailed software and hardware evaluation with physical memory synthesis, we demonstrate significant gains in area and power over existing approaches. Our work advances pebbling theory, and broadly, theoretical research for resource-efficient hardware design.

8 Acknowledgments

We thank Jim Aspnes, Anna Gilbert, and Ole Richter for their feedback and guidance at different stages of this work. This work is supported in part by grant CNS-2112562 from the NSF, and a Computing Innovation Fellowship from the CRA for Raghavendra Pradyumna Pothukuchi (under NSF grant 2127309).

References

[1] A Bolu Ajiboye, Francis R Willett, Daniel R Young, William D Memberg, Brian A Murphy, Jonathan P Miller, Benjamin L Walter, Jennifer A Sweet, Harry A Hoyen, Michael W Keith, P Hunter Peckham, John D Simeral, John P Donoghue, Leigh R Hochberg, and Robert F Kirsch. 2017. Restoration of reaching and grasping movements through brain-controlled muscle stimulation in a person with tetraplegia: a proof-of-concept demonstration. *The Lancet* 389, 10081 (May 2017), 1821–1830. doi:10.1016/s0140-6736(17)30601-3

[2] Samira Ataei and Rajit Manohar. 2019. AMC: An Asynchronous Memory Compiler. In *2019 25th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*. IEEE. doi:10.1109/async.2019.00009

[3] Charles H Bennett. 1989. Time/space trade-offs for reversible computation. *SIAM J. Comput.* 18, 4 (1989), 766–776.

[4] Jeremiah Blocki, Blake Holman, and Seunghoon Lee. 2022. The parallel reversible pebbling game: Analyzing the post-quantum security of iMHFs. In *Theory of Cryptography Conference*. Springer, 52–79.

[5] Jeremiah Blocki, Blake Holman, and Seunghoon Lee. 2024. The Impact of Reversibility on Parallel Pebbling. *Cryptology ePrint Archive* (2024).

[6] Jeremiah Blocki, Ling Ren, and Samson Zhou. 2018. Bandwidth-hard functions: Reductions and lower bounds. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1820–1836.

[7] Toni Böhnlein, Pál András Papp, and Albert-Jan N Yzelman. 2024. Brief Announcement: Red-Blue Pebbling with Multiple Processors: Time, Communication and Memory Trade-offs. In *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. 285–287.

[8] Siu Man Chan. 2013. Just a pebble game. In *2013 IEEE Conference on Computational Complexity*. IEEE, 133–143.

[9] Siu Man Chan, Massimo Lauria, Jakob Nordstrom, and Marc Vinyals. 2015. Hardness of approximation in PSPACE and separation results for pebble games. In *2015 IEEE 56th Annual Symposium on Foundations of Computer Science*. IEEE, 466–485.

[10] Xiaoming Chen, Yinhe Han, and Yu Wang. 2020. Communication lower bound in convolution accelerators. In *HPCA*.

[11] Stephen Cook and Ravi Sethi. 1974. Storage requirements for deterministic/polynomial time recognizable languages. In *Proceedings of the sixth annual ACM symposium on Theory of computing*. 33–39.

[12] Anuj Dawar and Bjarki Holm. 2012. Pebble games with algebraic rules. In *International Colloquium on Automata, Languages, and Programming*. Springer, 251–262.

[13] Erik D Demaine and Quanquan C Liu. 2017. Inapproximability of the standard pebble game and hard to pebble graphs. In *Workshop on Algorithms and Data Structures*. Springer, 313–324.

[14] Erik D Demaine and Quanquan C Liu. 2018. Red-blue pebble game: Complexity of computing the trade-off between cache size and memory transfers. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*. 195–204.

[15] Phillip Demarest, Nabi Rustamov, James Swift, Tao Xie, Markus Adamek, Hohyun Cho, Elizabeth Wilson, Zhuangyu Han, Alexander Belsten, Nicholas Luczak, Peter Brunner, Simon Haroutounian, and Eric C. Leuthardt. 2024. A novel theta-controlled vibrotactile brain–computer interface to treat chronic pain: a pilot study. *Scientific Reports* 14, 1 (Feb. 2024). doi:10.1038/s41598-024-53261-3

[16] Thaddeus Dryja, Quanquan C Liu, and Sunoo Park. 2018. Static-memory-hard functions, and modeling the cost of space vs. time. In *Theory of Cryptography: 16th International Conference, TCC 2018, Panaji, India, November 11–14, 2018, Proceedings, Part I 16*. Springer, 33–66.

[17] Patrick W Dymond and Martin Tompa. 1983. Speedups of deterministic machines by synchronous parallel machines. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*. 336–343.

[18] Jay L. Gill, Julia A. Schneiders, Matthias Stangl, Zahra M. Aghajan, Mauricio Vallejo, Sonja Hiller, Uros Topalovic, Cory S. Inman, Diane Villaroman, Ausaf Bari, Avishek Adhikari, Vikram R. Rao, Michael S. Fanselow, Michelle G. Craske, Scott E. Krahl, James W. Y. Chen, Merit Vick, Nicholas R. Hasulak, Jonathan C. Kao, Ralph J. Koek, Nanthia Suthana, and Jean-Philippe Langevin. 2023. A pilot study of closed-loop neuromodulation for treatment-resistant post-traumatic stress disorder. *Nature Communications* 14, 1 (May 2023). doi:10.1038/s41467-023-38712-1

[19] Martin Grohe and Martin Otto. 2015. Pebble games and linear equations. *The Journal of Symbolic Logic* 80, 3 (2015), 797–844.

[20] Qijing Huang, Po-An Tsai, Joel S. Emer, and Angshuman Parashar. 2024. Mind the Gap: Attainable Data Movement and Operational Intensity Bounds for Tensor Algorithms. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 150–166. doi:10.1109/isca59077.2024.00021

[21] Akifumi Imanishi, Zijian Xu, Masayuki Takagi, Sixue Wang, and Emilio Castillo. 2024. A fast heuristic to optimize time-space tradeoff for large models. *Advances in Neural Information Processing Systems* 36 (2024).

[22] Hong Jia-Wei and H. T. Kung. 1981. I/O complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing - STOC '81 (STOC '81)*. ACM Press. doi:10.1145/800076.802486

[23] Ce Jin, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua R. Wang. 2023. New Tools for Peak Memory Scheduling. arXiv:2312.13526 [cs.DS]

[24] Ioannis Karageorgos, Karthik Sriram, Ján Veselỳ, Michael Wu, Marc Powell, David Borton, Rajit Manohar, and Abhishek Bhattacharjee. 2020. Hardware-software co-design for brain-computer interfaces. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 391–404.

[25] Sohee Kim, Prashant Tathireddy, Richard A. Normann, and Florian Solzbacher. 2007. Thermal Impact of an Active 3-D Microelectrode Array Implanted in the Brain. *IEEE Transactions on Neural Systems and Rehabilitation Engineering* 15, 4 (Dec. 2007), 493–501. doi:10.1109/tnsre.2007.908429

[26] Ravi Kumar, Manish Purohit, Zoya Svitkina, Erik Vee, and Joshua Wang. 2019. Efficient rematerialization for deep networks. *Advances in Neural Information Processing Systems* 32 (2019).

[27] Grzegorz Kwasniewski, Tal Ben-Nun, Lukas Gianinazzi, Alexandru Calotoiu, Timo Schneider, Alexandros Nikolaos Ziogas, Maciej Besta, and Torsten Hoefler. 2021. Pebbles, graphs, and a pinch of combinatorics: Towards tight I/O lower bounds for statically analyzable programs. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 328–339.

[28] Grzegorz Kwasniewski, Marko Kabic, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Jens Eirik Saethre, André Gaillard, Timo Schneider, Maciej Besta, Anton Kozhevnikov, Joost VandeVondele, et al. 2021. On the parallel i/o optimality of linear algebra kernels: Near-optimal matrix factorizations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[29] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. 2019. Red-blue pebbling revisited: near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–22.

[30] Mikhail A. Lebedev and Miguel A. L. Nicolelis. 2017. Brain-Machine Interfaces: From Basic Science to Neuroprostheses and Neurorehabilitation. *Physiological Reviews* 97, 2 (April 2017), 767–837. doi:10.1152/physrev.00027.2016

[31] Quanquan Catherine Liu. 2017. Red-blue and standard pebble games: Complexity and applications in the sequential and parallel models.

[32] Jakob Nordstrom. 2013. Pebble games, proof complexity, and time-space trade-offs. *Logical Methods in Computer Science* 9 (2013).

[33] Carina R Oehrn, Stephanie Cernera, Lauren H Hammer, Maria Shcherbakova, Jiaang Yao, Amelia Hahn, Sarah Wang, Jill L Ostrem, Simon Little, and Philip A Starr. 2023. Personalized chronic adaptive deep brain stimulation outperforms conventional stimulation in Parkinson's disease. (Aug. 2023). doi:10.1101/2023.08.03.23293450

[34] Auguste Olivry, Guillaume Iooss, Nicolas Tollenaere, Atanas Rountev, P. Sadayappan, and Fabrice Rastello. 2021. IOOpt: automatic derivation of I/O complexity bounds for affine programs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21)*. ACM, 1187–1202. doi:10.1145/3453483.3454103

[35] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P. Sadayappan, and Fabrice Rastello. 2020. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM, 808–822. doi:10.1145/3385412.3385989

[36] Pál András Papp and Roger Wattenhofer. 2020. On the Hardness of Red-Blue Pebble Games. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20)*. ACM. doi:10.1145/3350755.3400278

[37] Michael S Paterson and Carl E Hewitt. 1970. Comparative schematology. In *Record of the Project MAC conference on concurrent systems and parallel computation*. 119–127.

[38] Arend-Jan Quist and Alfons Laarman. 2023. Optimizing quantum space using spooky pebble games. In *International Conference on Reversible Computation*. Springer, 134–149.

[39] Ravi Sethi. 1973. Complete register allocation problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*. 182–195.

[40] Ravi Sethi. 1982. Pebble games for studying storage sharing. *Theoretical Computer Science* 19, 1 (1982), 69–84.

[41] Karthik Sriram, Raghavendra Pradyumna Pothukuchi, Michal Gerasimiuk, Muhammed Ugur, Oliver Ye, Rajit Manohar, Anurag Khandelwal, and Abhishek Bhattacharjee. 2023. SCALO: An Accelerator-Rich Distributed System for Scalable Brain-Computer Interfacing. In *2023 ACM/IEEE 50th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[42] Yuya Uezato. 2021. Accelerating XOR-based erasure coding using program optimization techniques. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.

[43] Muhammed Ugur, Raghavendra Pradyumna Pothukuchi, and Abhishek Bhattacharjee. 2024. Swapping-Centric Neural Recording Systems. (2024). doi:10.48550/ARXIV.2409.17541

[44] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures: dealing with massive data. *Comput. Surveys* 33, 2 (June 2001), 209–271. doi:10.1145/384192.384193

[45] SA Walker. 1971. *Some graph games related to the efficient calculation of expressions*. IBM Thomas J. Watson Research Center.

[46] Neil Weste and David Harris. 2010. *CMOS VLSI Design: A Circuits and Systems Perspective* (4th ed.). Addison-Wesley Publishing Company, USA.