MIRAGE: MILP-Based Block Grouping for Real-Time Signal Processing

Tiancheng He
Department of Computer Science
Vanderbilt University
Nashville, U.S
tiancheng.he@vanderbilt.edu

Bryan C. Ward
Department of Computer Science
Vanderbilt University
Nashville, TN, U.S
bryan.ward@vanderbilt.edu

***Abstract*—High-frequency signal-processing applications such as software-defined radio (SDR) pose unique challenges for real-time scheduling. Unlike traditional embedded workloads with relatively modest invocation rates, SDR pipelines operate at kilohertz to gigahertz frequencies, where the overhead of frequent task invocations—including context switching, cache reloading, and synchronization—can dominate execution time. As a result, software platforms such as GNU Radio leverage heuristic-based scheduling to minimize these overheads to improve throughput. Unfortunately, this provides little predictability and no analytical guarantees.**

This paper introduces MIRAGE (MILP-based Real-time Assignment and Grouping for EDF), a framework that formulates block grouping and batching as an optimization problem, where the decision variables determine how tasks are clustered into groups. Applications are modeled as Processing Graph Method (PGM) task graphs, which naturally capture the structure of SDR pipelines. Building on prior work that identified the marginal cost of task initialization as the dominant overhead, this paper combines two complementary strategies: *intra-block batching*, which amortizes initialization within tasks, and *inter-block merging*, which reduces redundant overheads across consecutive tasks. This paper formulates the grouping problem as a mixed-integer linear program (MILP) that minimizes utilization and end-to-end latency while preserving schedulability under global earliest-deadline-first (G-EDF) scheduling. The evaluations demonstrate that MIRAGE consistently reduces invocation overhead, improves cache efficiency, and achieves significant reductions in per-sample execution time. Importantly, the MILP solver converges rapidly, yielding high-quality solutions suitable for use in real-time contexts. These results show that optimization-driven grouping offers a systematic and analytically grounded path toward predictable, high-performance SDR execution on multi-core platforms.

I. Introduction

Software-defined Radio (SDR) is an increasingly important technology in which radio signals are processed in software rather than dedicated hardware. This shift enables flexible, adaptable, and upgradeable wireless communication: by updating software or running different algorithms, performance can be improved, protocols can be updated, or communication can be shifted to different spectrum bands. SDR has thus become a key enabler in domains such as cyber-physical systems, IoT devices, cellular and satellite networks, and even military applications, where it supports dynamic and resilient communication in spectrum-contested environments.

In SDR systems, signal-processing pipelines are commonly represented as flow-graphs, in which data samples pass through a chain of computational blocks. In hardware-based designs, these blocks are directly implemented in circuits that operate in parallel with predictable performance. In contrast, SDR flow-graphs execute on general-purpose CPUs, where blocks compete for processing time, experience cache interference, and operate under limited parallelism. As a result, SDR implementations often face significant jitter and end-to-end latency, which pose barriers to their use in real-time settings.

The root of these challenges lies in how high-frequency workloads are scheduled. Many existing real-time graph scheduling models assume invocation rates that are relatively modest (e.g., tens to hundreds of Hz, as in video or control applications). In SDR, the situation is radically different: signals are sampled at kilohertz to gigahertz frequencies (e.g., 44.1 kHz audio, 2.4 GHz Wi-Fi) or faster, which forces flow-graphs to process large batches of samples with very short per-sample execution budgets. At these rates, even small scheduling inefficiencies can accumulate into significant jitter and latency. Popular SDR frameworks such as GNU Radio adopt pragmatic heuristics to cope with this problem: each block is assigned its own thread, and execution is triggered only when its input buffer is partially filled. This reduces context switches and improves cache affinity, but it also introduces variability in invocation sizes and timing, leaving scheduling decisions to the operating system's general-purpose scheduler. The resulting unpredictability makes it difficult to reason about worst-case performance, and thus prevents rigorous real-time analysis.

Prior work introduced the *Marginal Cost Model* [1] to formalize the benefits of batching within a block. For clarity, we henceforth refer to this as the *intra-block marginal cost model*. The model captures the fact that execution time can be expressed as fixed initialization cost (I_v) per invocation plus the marginal (Δ_v) cost per processed sample. Since $I_v \gg \Delta_v$ for typical signal-processing blocks, batching more samples per invocation amortizes initialization costs, reduces context switches, and improves predictability. Experiments in [1] demonstrated that this *intra-block batching* strategy significantly improves utilization and predictability in SDR pipelines.

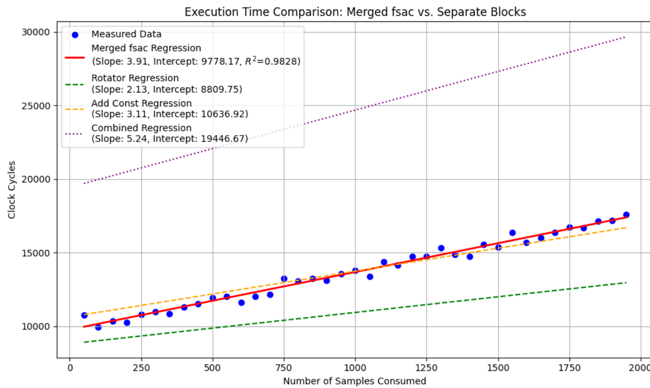

Fig. 1: Execution time comparison of merged block vs. separate `rotator` and `add_const` blocks. Inter-block merging reduces redundant initialization overheads and improves efficiency.

Motivated by this same model, we extend the idea to explore *inter-block batching*, where multiple adjacent blocks are executed together as a single unit. To evaluate this approach, we profiled four configurations: (1) frequency shifting alone, (2) add-constant alone, (3) the sum of their separate costs, and (4) a merged implementation that combines frequency shifting and add-constant into one block. As shown in Fig. 1, the merged configuration not only reduces the one-time initialization overhead but also shows evidence of secondary benefits—what we call an *inter-block marginal cost* improvement—arising from fewer context switches and improved cache reuse. These findings suggest that inter-block batching can complement intra-block batching, further reducing overhead and enabling more predictable execution in SDR pipelines.

Taken together, these two results—that intra-block batching and inter-block batching are both beneficial—raise a fundamental question: *how should one decide the batching factor for each block, and which blocks should be grouped together*? Ad hoc heuristics cannot easily answer this, since the tradeoffs depend on block-specific features such as execution cost, data rates, and communication patterns. This motivates the use of optimization-based methods to systematically explore the design space of batching and grouping decisions.

Mixed-Integer Linear Programming (MILP) is a particularly powerful tool in this setting: it naturally models both discrete decisions (e.g., whether tasks are grouped together) and continuous variables (e.g., execution times), and it allows tradeoffs between schedulability, latency, and overhead to be explored systematically. Rather than relying on ad hoc heuristics, optimization-based approaches can provide predictable guarantees while simultaneously improving efficiency.

In this paper, we present **MIRAGE** (MILP-based Real-time Assignment and Grouping for EDF), a framework that applies optimization to the problem of task grouping and batching in real-time SDR applications. MIRAGE represents applications as Processing Graph Method (PGM) task graphs [2] and formulates grouping and batching decisions as a MILP under global earliest-deadline-first (G-EDF) scheduling. In this work, we focus on the soft real-time (SRT) scheduling of processing graphs. For signal-processing systems, the primary requirement is that processing keeps pace with the incoming signal rate while providing bounded latency on the output, rather than guaranteeing that every individual deadline is always met. Our evaluation shows that MIRAGE can substantially reduce invocation overheads and jitter while converging quickly to high-quality solutions. These results demonstrate that optimization-based design provides a systematic way to bring predictability to high-frequency SDR workloads.

Contributions

The main contributions of this paper are:

- **MILP-based grouping:** Formulate SDR block grouping as a MILP that captures schedulability constraints, utilization bounds, and communication locality.
- **Unified batching:** Extend the marginal cost model to jointly consider intra-block batching and inter-block merging, systematically reducing invocation overhead at multiple levels.
- **Schedulability-aware design and evaluation:** Our framework preserves acyclicity, enforces utilization limits, and ensures G-EDF compatibility. Experiments on hundreds of synthetic PGM graphs show consistent reductions in utilization and latency, with the MILP solver converging quickly to practical solutions.

II. Background

In many embedded systems and real-time systems, complicated data processing is modeled to be a directed acyclic graph (DAG). In a DAG, vertices represent tasks while edges represent the precedence relationship between vertices/tasks. The reason why such modeling is favorable is that we can benefit from parallel execution and targeted scheduling on different architectures.

Next we review the PGM framework originally designed for signal-processing applications [3]. Liu and Anderson showed that a PGM graph could be scheduled by a global earliest-deadline-first-like (G-EDF-like) scheduler and does not suffer from utilization loss on a multi-core platform, while ensuring bounded latency.[1] This correctness condition is particularly well-suited for signal processing, where strict deadlines are often less critical than maximizing supported frequency or bandwidth—ideally with minimal latency.

To demonstrate bounded deadline latency under G-EDF-like scheduling, Liu and Anderson [4] transform the PGM into a set of sporadic tasks. Once in this form, existing results from the literature on sporadic task systems [5]–[7] can be applied to establish bounds on latency. This transformation also allows the use of utilization and latency as performance metrics, both of which can be computed directly from the sporadic task set representation.

[1]In the original work by Liu and Anderson, the term tardiness was used. Since deadlines in PGM graphs are fixed offsets from release times, bounded tardiness directly implies bounded latency.

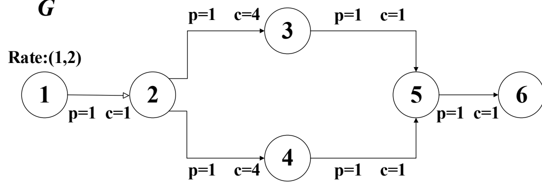

Fig. 2: Example PGM Graph G

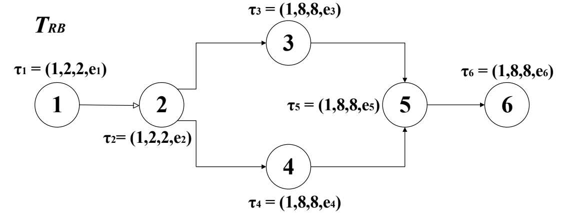

Fig. 3: The RB-parameters, *RB(x, y, d, e)*, for each task in G

A. Processing graph method

A PGM graph, G, consists of nodes that are connected by directed edges. One node corresponds to a signal-processing task while one edge corresponds to a first-in, first-out (FIFO) queue denoting the data passage between successive tasks. Due to PGM graphs' acyclic nature, there will be at least one *source node* for any PGM graph. *Source node* is a node with only outgoing edges. Symmetrically, a PGM graph would have at least one *sink node*, which has no incoming edge. Edges in PGM graphs are also specified in three aspects: produce amount, threshold and consume amount. Assume that there is an edge from a node i to a node j. The produce amount $p_{i \to j}$ denotes the amount of data tokens that are fed into the queue every time i executes. j will consume $c_{i \to j}$ data tokens from the queue. Fig.2 shows an example of PGM graphs.

B. Rate-based task model

To convert a PGM graph G into a set of schedule tasks, [2] and [3] presented analyses that derive *execution rates* for all nodes in G. The execution rate (x, y) means that a task would execute x times in any time interval y. An invocation of a task T is called a job τ and a task's execution cost e denotes the processor time it needs to execute. A task's execution rate and execution cost together are used to define a task in a Rate-Based (RB) Task Model. Here we use T_{RB} to denote the RB Task Model and G for the PGM graph that the T_{RB} originates from. As proven in [2], the RB-parameters of any task except the source task can be calculated using Lemma II.1

Lemma II.1. *For any non-source task τ_u in T_{RB}, let V denote the set of predecessor tasks of τ_u. For any τ_v in V, let $Rate_v = (x_v, y_v)$ be a valid execution rate. Then, the execution rate $Rate_u = (x_u, y_u)$ is valid for τ_u if*

$$y_u = \operatorname{lcm}\left\{ \frac{c_{v \to u} \cdot y_v}{\gcd\left(p_{v \to u} \cdot x_v, c_{v \to u}\right)} \middle| \forall v \in V \right\}$$
$$x_u = y_u \cdot \frac{p_{v \to u}}{c_{v \to u}} \cdot \frac{x_v}{y_v}, \exists v \in V$$

Following the formulation in [2], [3], we assign each rate-based (RB) task τ_u a relative deadline defined as $d_u = y_u / x_u$. We impose a monotonic rate constraint on the precedence graph G, such that for every directed edge from node u to $v \in$ G, it holds that $d_u \leq d_v$. This condition ensures that the execution rate does not increase along any path in the PGM graph. We assume the existence of a single source node with a fixed periodic execution rate, i.e., $x = 1$, which is derived from the application's sampling period. Additionally, we assume that the execution cost e_u for each task τ_u is given. Throughout this paper, we represent each RB task using the tuple $\tau_u = (x_u, y_u, d_u, e_u)$, where x_u and y_u denote the production and consumption rates, respectively, d_u is the relative deadline, and e_u is the execution cost.

C. Utilization

After converting the PGM graph into a sporadic task set, T_S, we can calculate the utilization of each task $u_v = e_v / p_v$. Utilization describes the fraction of total processor time a task τ_v requires. We call a soft real-time system with G-EDF-like scheduler schedulable on M processors if the sum of all tasks' utilization is under M.

D. Latency

For signal-processing pipelines, having bounded tardiness ensures that latency does not grow unbounded as the program proceeds. In this work, the end-to-end latency of a task set is defined as the maximum time between the arrival of data at the source node and the completion of processing at the sink node.

Formally, following the analysis in [1], if T_S is SRT schedulable under G-EDF on M processors, then the total latency L along a path from a source node j to a sink node w can be upper bounded as:

$$L \leq \big(\max(\{F_q \mid q \in j \to w\}) - 1 \big) \cdot pd_j + \sum_{i \in q} r_i, \quad (1)$$

where F_q is the maximum number of times that τ_j must execute before τ_w can start, q is the path from j to w that maximizes this bound, pd is the period of the source task, and r_i is the worst-case response time of task τ_i on path q.

E. The Marginal Cost Model

As established in prior work [1], the execution time of signal-processing blocks in software-defined radio (SDR) systems is composed of two dominant components: a high one-time cost incurred at the start of execution, and a much smaller per-sample cost. This observation forms the basis of the *Marginal Cost Model.*

Definition II.1 (Marginal Cost Model [1])**.** *Let $\tau_v = (p_v, d_v, e_v)$ be a sporadic task, where e_v denotes its execution cost. Let I_v be the* initialization cost *representing time spent on*

cache warming, context setup, and thread synchronization. Let Δ_v *be the* marginal cost per sample*, and let* c_v *be the number of samples consumed per invocation. Then, the execution cost* e_v *is modeled as:*

$$e_v = I_v + \Delta_v \cdot c_v \quad (2)$$

This linear cost model is supported by profiling results that reveal a sharp difference between the fixed overhead of starting a block and the negligible cost of processing each additional sample. In all cases studied in [1], the initialization cost I_v dominates the marginal cost Δ_v, i.e., $I_v \gg \Delta_v$. This disparity motivates batching techniques that reduce the frequency of high-overhead invocations by processing more samples per execution, thereby improving cache locality and reducing latency and jitter in high-frequency SDR applications.

F. Rate-Exploiting Batching

As established in [1], batching is an effective strategy for reducing utilization in high-frequency signal-processing graphs by amortizing initialization costs across multiple samples. In particular, *Rate-Exploiting Batching* takes advantage of imbalances in produce/consume ratios across edges in a PGM representation to identify opportunities where upstream nodes can be executed less frequently, but with larger batches.

To formalize this, consider a task τ_v with multiple outgoing edges. Each edge is characterized by a ratio between its produce and consume parameters. If all outgoing edges share the same ratio, and this ratio is an integer, then the node τ_v can be safely executed in larger batches. Specifically, the batch factor is given by the greatest common divisor (GCD) of these ratios. After applying this transformation, the consume value on every incoming edge to τ_v is multiplied by the batch factor, and the produce value on every outgoing edge from τ_v is likewise scaled. This ensures semantic correctness while increasing the amount of data processed per invocation, thereby reducing the total number of invocations and amortizing the fixed initialization cost.

Rate-Exploiting Batching is applied in a reverse topological traversal of the graph. A node is eligible for batching only if all its outgoing edges satisfy the criteria for integral scaling. Importantly, this transformation does not require modifying the rates of predecessor nodes, and thus has only minimal impact on overall system latency.

Here we revisit the example PGM graph in Fig. 2. In this graph, Node 2 produces data for both Node 3 and Node 4. However, due to the mismatch between the produce and consume values on its outgoing edges, Node 2 must execute four times before either Node 3 or Node 4 has enough data to execute once. This leads to four separate invocations of Node 2, each incurring its own initialization cost.

With *Rate-Exploiting Batching*, we can consolidate these four invocations of Node 2 into a single batched execution, thereby paying the initialization cost only once. This reduces the total number of invocations while preserving the dataflow semantics of the graph. Fig. 4 illustrates the batched version of the original graph, where the consume/produce parameters

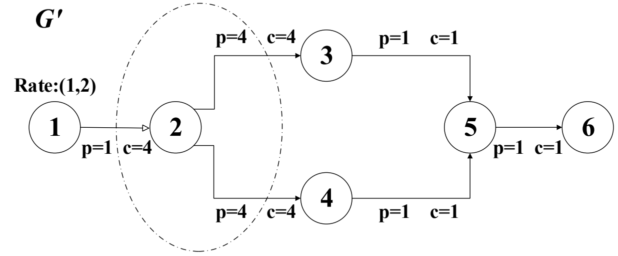

Fig. 4: Example of applying rate-exploiting batching to a PGM graph. Batching reduces invocation frequency by scaling consume/produce values

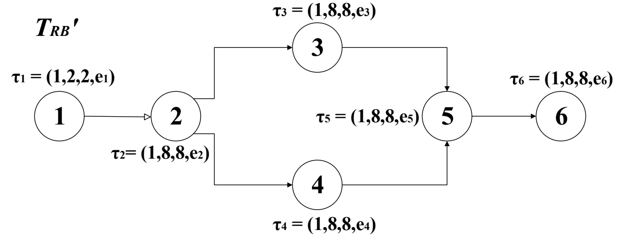

Fig. 5: Rate-based (RB) parameters $RB(x, y, d, e)$ for each task in the PGM graph after applying rate-exploiting batching.

have been scaled to reflect the batched execution. Correspondingly, Fig. 5 shows the updated Rate-Based (RB) task parameters, demonstrating how batching reduces invocation frequency while maintaining correctness.

In this project, Rate-Exploiting Batching is applied as a pre-processing step, after which MILP-based optimization is performed on the transformed graph to determine how best to group tasks into executable partitions while preserving dependency constraints. This separation allows us to decouple rate optimization from global scheduling, resulting in both utilization improvements and analytical tractability.

III. MILP-Based Grouping Model

In this section, we develop the optimization framework that underpins MIRAGE. We begin by describing how nodes in a PGM representation can be consolidated to reduce redundant overheads. We then formalize the semantics of grouping tasks into co-scheduled units, followed by our Mixed-Integer Linear Programming (MILP) formulation that captures these design choices under real-time constraints.

A. PGM Graph Node Merging

To improve the predictability and efficiency of high-frequency signal processing pipelines, we introduce a graph-level transformation called *node merging*. This concept should be understood prior to our MILP-based optimization model, as it provides the structural foundation for grouping tasks and reducing overhead.

Following the Rate-Exploiting Batching pass, nodes in the PGM graph tend to exhibit a **clustering effect**, where many nodes converge to a small set of identical or similar execution

rates. This phenomenon, combined with the insight from the marginal cost model (see Eq. 2) and empirical block-merging experiments in Section I, motivates a strategy to consolidate nodes with shared rates into single execution units.

Since context-switch overhead and cache reloading penalties are amortized when tasks are executed in a shared thread, merging tasks with the same rate can significantly reduce per-sample execution cost. In GNU Radio terms, this resembles collapsing multiple blocks into one custom block to reduce per-invocation overhead.

We define the node merging process as follows.

Execution Cost: The execution cost e_g of a merge group g is computed as:

$$e_g = I_g + \sum_{i \in g} \Delta_i \cdot c_i$$

where I_g is the initialization cost incurred once per merged node (not per task), and Δ_i and c_i are the marginal cost and consumption of each task i in the group. This formulation reflects amortized overhead and follows the marginal cost model [1]. Furthermore, our experiment in Fig. 1 demonstrates that merging consecutive blocks further reduces execution cost. Specifically, by merging blocks into one merged implementation, we observed that the combined regression line has a lower effective slope than the sum of the two separate regressions. This shows that inter-block merging not only amortizes initialization costs but also provides secondary savings due to reduced context switches and improved cache locality.

Edge Construction: A merge group g_i connects to another merge group g_j if there is any dependency between tasks in g_i and g_j. The new edge's produce and consume values are obtained by summing over the corresponding values of all inter-task edges:

$$p_{g_i \to g_j} = \sum_{(u,v) \in E_{ij}} p_{u \to v}, \quad c_{g_i \to g_j} = \sum_{(u,v) \in E_{ij}} c_{u \to v}$$

where E_{ij} is the set of all edges from tasks $u \in g_i$ to $v \in g_j$ in the original PGM.

Rate Parameters: The new execution rate (x_H, y_H) for a merged node is derived using Lemma II.1, treating merged groups as atomic units connected by the newly aggregated edges.

Fig. 6 illustrates this merging transformation. In the original graph (top), three upstream nodes (1, 2, and 3) each produce to node 4 with identical data rates and consume values. These can be safely merged into a single composite node (bottom), which consolidates their execution into a single task with proportionally scaled dataflow parameters.

This transformation reduces the total number of invocations and improves efficiency. Building on this foundation, we next formalize the semantics of grouping tasks and the scheduling implications of executing them together.

B. MILP Formulation

We formulate the grouping problem as a *Mixed-Integer Linear Program (MILP)* with the following decision variables:

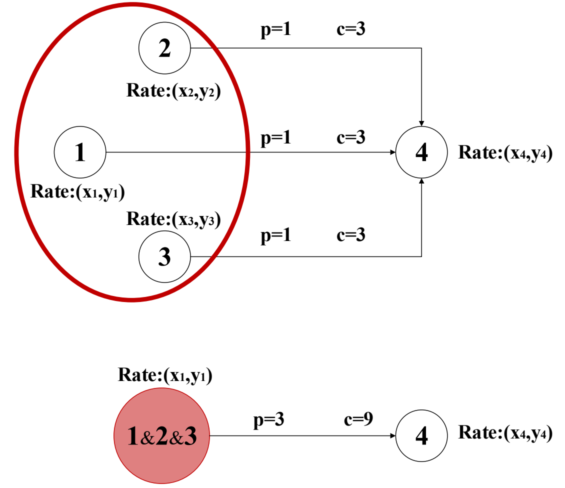

Fig. 6: Example of node merging: tasks 1, 2, and 3 with the same execution rate are merged into a single composite node.

- $N[u][m] \in \{0, 1\}$: *Node in Group Variable* that equals 1 if T_m is assigned to group g_u, and 0 otherwise.
- $D[i][j] \in \{0, 1\}$: *Group Dependency Variable* that equals 1 if there exists a dependency from any task in group g_i to any task in group g_j.
- $h_{u,v}^{m,n} \in \{0, 1\}$: *Auxiliary co-location variable* that equals 1 iff task m is assigned to group u and task n is assigned to group v. These variables are used to linearize products of binary variables in the objective and dependency constraints.
- $z_{i,j} \in \{0, 1\}$: *Acyclicity selector* used in the big-M formulation that prevents mutual dependencies between groups i and j.

C. Objective Function

To reduce context-switch overhead and improve cache locality, we aim to maximize *intra-group communication* by assigning directly dependent tasks to the same group. In other words, tasks connected by dataflow edges are preferentially grouped together. For each pair of tasks (T_m, T_n) within a group g, let non-variable t_{mn} denote the amount of data transferred from T_m to T_n. We initialize the number of groups to be equal to the number of tasks, k, prior to merging, i.e., $G = \{T_1, T_2, \ldots, T_k\}$. Note that some groups may remain empty. The optimization objective is defined as:

$$\max \sum_{u=1}^{k} \sum_{(m,n)} t_{mn} \cdot N[u][m] \cdot N[u][n] \tag{3}$$

Here, $N[u][m]$ denotes the binary *Node-in-Group Variable*, which is equal to 1 if task T_m is assigned to group g_u (analogously, $N[u][n]$ indicates whether task T_n belongs to group g_u). The product $N[u][m] \cdot N[u][n]$ ensures that communication volume t_{mn} contributes to the objective only when both tasks are assigned to the same group.

In the MILP implementation, we linearize Eq. (3) by using $h_{u,u}^{m,n}$ to represent the product $N[u][m] \cdot N[u][n]$ and adding the standard linearization constraints:

$$h_{u,u}^{m,n} \leq N[u][m], \tag{4}$$
$$h_{u,u}^{m,n} \leq N[u][n], \tag{5}$$
$$h_{u,u}^{m,n} \geq N[u][m] + N[u][n] - 1, \tag{6}$$

The objective then becomes a purely linear expression:

$$\max \sum_{u=1}^{k} \sum_{(m,n)} t_{mn} \cdot h_{u,u}^{m,n}. \tag{7}$$

D. Constraints

To ensure that the solution is valid and schedulable under G-EDF, we impose the following constraints:

Unique Group Assignment: Each task T_m must belong to exactly one group:

$$\sum_{u=1}^{k} N[u][m] = 1 \quad \forall m \in \{1, \ldots, k\} \tag{8}$$

Group Utilization Constraint: The total execution cost of all tasks assigned to a group g_u must not exceed the maximum group capacity U_{max}, a tunable parameter in $[0, 1]$. We make U_{max} tunable in our experiments to evaluate the effectiveness of MIRAGE: smaller group capacities restrict merging opportunities, thereby reducing MIRAGE's impact, whereas larger capacities enable more aggressive grouping and greater potential savings.

$$\sum_{m=1}^{k} e_m \cdot N[u][m] \leq U_{\text{max}} \quad \forall u \in \{1, \ldots, k\} \tag{9}$$

Acyclic Group Dependency: The grouped graph must remain acyclic to prevent deadlocks. If a group g_A contains a task that is the predecessor of another task in group g_B, then g_B must not contain a task that is the predecessor of a task in g_A. This applies to multi-group scenarios as well — meaning there cannot be a cycle such as g_X depends on g_Y, g_Y depends on g_Z, and simultaneously g_Z depends on g_X.

Let $\text{TC}_{m,n} \in \{0, 1\}$ denote the (fixed) transitive-closure matrix of the original PGM, i.e., $\text{TC}_{m,n} = 1$ iff there exists a path from task m to task n. We derive group-to-group dependencies as:

$$D_{i,j} = \sum_{m,n} h_{i,j}^{m,n} \cdot \text{TC}_{m,n}. \tag{10}$$

Thus, $D_{i,j}$ is not a free decision variable; it is completely determined by the group assignment and the original graph. Therefore, the MILP cannot remove or invent dependencies—all original edges are automatically preserved.

To prevent cyclic dependencies between groups, we enforce that for each unordered pair (i, j), dependencies may appear in

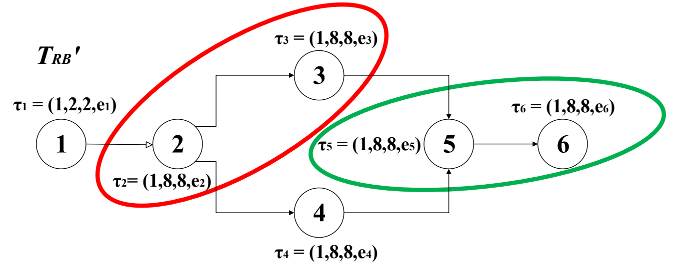

Fig. 7: PGM graph after MIRAGE grouping: nodes are partitioned into color-coded groups, each representing a co-scheduled execution unit.

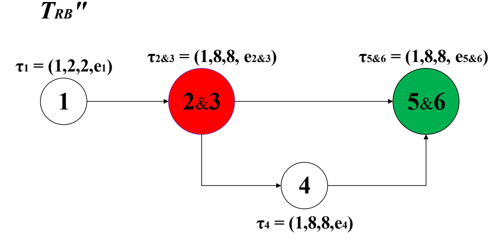

Fig. 8: MIRAGE-transformed PGM graph: grouped tasks are collapsed into single logical nodes, yielding a simplified execution structure.

at most one direction. Using a big-M formulation with binary selector $z_{i,j}$:

$$D_{i,j} \leq M \cdot z_{i,j}, \tag{11}$$
$$D_{j,i} \leq M \cdot (1 - z_{i,j}), \tag{12}$$

for all $i \neq j$. This ensures that if $D_{i,j} > 0$ then $D_{j,i} = 0$, and conversely, which eliminates 2-cycles and, combined with the transitive closure, prevents longer cycles.

Implementation Note: For clarity of presentation, we use products of binary variables in the paper. In the actual MILP, all such products are replaced by auxiliary variables and linear constraints as shown above. Modern MILP solvers accept quadratic terms, but we intentionally keep our formulation purely linear for efficiency.

E. MIRAGE Output

Once the MILP-based grouping model is solved using Gurobi, MIRAGE produces a mapping from each original task node to a group identifier. This grouping reflects an optimized partitioning of the PGM graph that balances utilization constraints while maximizing intra-group communication.

Fig. 7 illustrates the result on the example graph from Fig. 3. In Fig. 7, tasks are color-coded according to their assigned groups, with each color representing a co-scheduled execution unit. MIRAGE then merges each group into a merged node, yielding the simplified graph shown in Fig. 8. The transformed structure reduces the number of execution units, amortizes initialization costs across merged tasks, and exposes a clearer execution flow suitable for schedulability analysis.

TABLE I: Experiment Parameters

(a) Top-level Parameters for PGM Generation

Batch Size	$[1, 2, 4, 6, 8, 12, 16]$
Graph Size	{Light, Heavy}

(b) Graph Size Configuration

Graph Size	Light	Heavy
Num. Nodes	$[5, 6, \ldots, 15]$	$[15, 16, \ldots, 25]$
Branching Degree	$[1, 2, 3]$	$[1, 2, 3, 4]$

(c) Block Cost Distributions for Initialization (I) and Marginal Cost (Δ)

Block Type	Probability	I (μs)	Δ (μs)
1:1	0.5	[3,4,5]	0.001
1:Uniform[2–10]	0.5	[6,7,8]	0.005

(d) Group Utilization Limit Categories

Utilization Limit	$[0.1, 0.33, 0.66, 1]$

IV. Evaluation

In this section, we evaluate MIRAGE with respect to two primary metrics: total utilization and end-to-end latency. Our experiments use randomly generated PGM graphs to model a wide range of workloads. After applying rate-exploiting batching as a preprocessing step, we solve the MILP-based grouping formulation using the `Gurobi` solver, which provides both high-quality and efficient converging solutions. We also investigated the effects of grouping. The results allow us to quantify the benefits of grouping and batching under G-EDF scheduling and to assess the trade-offs introduced by varying group utilization limits and batching factors.

A. Experiment Setup

Following the experimental methodology in [1], each synthetic PGM is generated as an upper-triangular adjacency matrix: vertices are assigned a fixed topological order $0, 1, \ldots, N-1$, and edges are added only from lower to higher indices, which guarantees acyclicity by construction. For each vertex, an outgoing degree is sampled from UniformInt$(1, 3)$, and we attempt to place edges to later vertices until this degree is satisfied. If a non-source vertex has no predecessor, a fallback edge from $(v-1)$ to v is inserted to maintain connectivity. Produce/consume rates (p, c) are then assigned using the PGM Level-I/II constraints or drawn from an SDR-motivated distribution when unconstrained. For each combination of parameters in Table Ia, 600 PGMs were generated with randomly selected values for the corresponding graph-size distributions shown in Table Ib. The rate of the source node is set to match a 1 MHz sampling rate. The produce and consume values are randomly generated so that every node has an equal chance to be a one-to-one function (e.g., frequency shift or constant multiply) or a decimating function (e.g., filter). We randomly select I and Δ from the distributions shown in Table Ic. To test the effect of the group utilization limit, we experimented with different limits as shown in Table Id. Since the utilization of a processor can be at most one, we set the maximum utilization for each group to be one.

After defining these parameters, we convert each generated PGM graph into a sporadic task set and evaluate total utilization and end-to-end latency bounds under G-EDF scheduling.

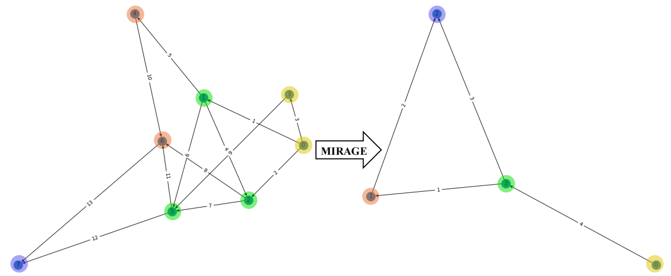

Fig. 9: PGM graph after MIRAGE grouping, with color-coded groups representing co-scheduled execution units

B. MIRAGE Results

Here we show how MIRAGE treats a randomly generated example PGM graph as shown in Fig. 9. For visibility, each different color represents different groups. Then we can create a new graph consisting of the grouped nodes. We use one node to represent a group, as is shown in the right graph.

With the MIRAGE-transformed PGM graph, we evaluate the impact of batching and grouping on utilization and latency. 'Utilization' or 'Latency' curves denotes the untreated PGMs; 'ReUtil' or 'RELate' denotes Rate-Exploiting Batching treated PGMs and "MIRAGE-$U_{\max}$" indicates MIRAGE applied with group utilization cap $U_{\max}$. Higher feasibility ratio indicates better schedulability.

The x-axis labeled "samples consumed" represents the batch size produced by the Rate-Exploiting Batching pass prior to applying MIRAGE. Larger values correspond to larger pre-MIRAGE batches. Varying this parameter allows us to examine how MIRAGE responds to different sample granularities and block workloads.

In Fig. 10, we show how total utilization and end-to-end latency evolves as the batching factor changes. Every data point represents the average utilization/end-to-end latency among the 100 randomly generated applications for the selected batch size.

Fig. 10 and Fig. 11 show the trend of total utilization as a function of the batch size for both light and heavy graphs. We first focus on the behavior of the MIRAGE configuration. In both light and heavy graphs, the MIRAGE lines demonstrate a clear and consistent downward trend as the batch size increases. This decline reflects the reduction in per-invocation overhead due to fewer total invocations, which improves cache reuse. Larger batch sizes allow tasks to be grouped more effectively, thereby amortizing the initialization cost across a larger number of samples. This confirms that cache-locality-driven grouping yields substantial improvements in processor utilization when more computation is done per invocation.

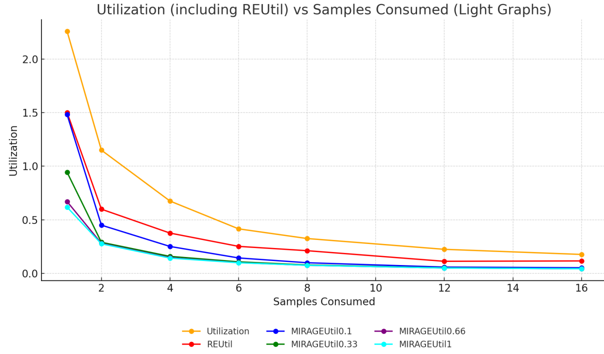

Fig. 10: Average utilization of light PGM graphs as a function of batch size

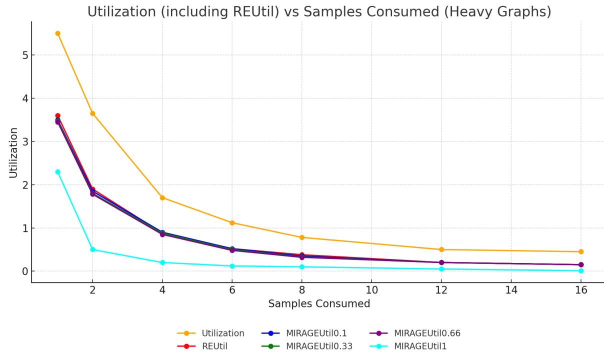

Fig. 11: Average utilization of heavy PGM graphs as a function of batch size

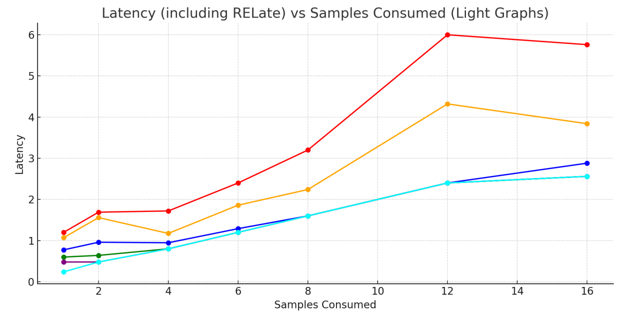

Fig. 12: Average latency of light PGM graphs as a function of batch size

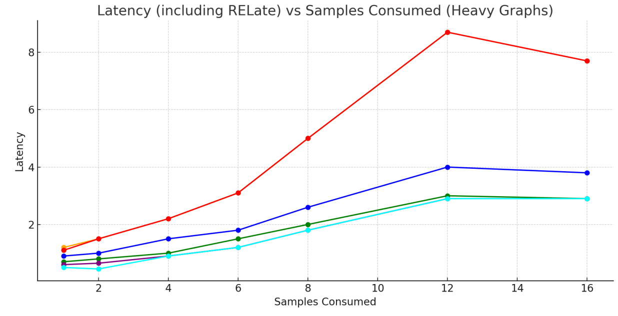

Fig. 13: Average latency of heavy PGM graphs as a function of batch size

C. Effect of Group Utilization Limit

We also investigate how the group utilization limit influences the effectiveness of MIRAGE. Recall that the group utilization limit imposes an upper bound on the cumulative utilization of tasks assigned to a single group. A larger utilization limit allows more tasks to be merged together into one group, enabling more opportunities for cache-locality enhancement and batching overhead amortization. In contrast, a smaller limit restricts group sizes, leading to more fragmented groupings with fewer tasks per group.

As shown in the Figs. 10, 11, 12, and 13, when the utilization limit is low (e.g., 0.1), the benefits of our MILP-based grouping diminish. In this case, the resulting utilization curves tend to approach the baseline established by the rate-exploiting (RE) technique alone. This is because the tight capacity constraint severely limits the solver's ability to cluster nodes with significant intra-group communication potential, thereby underutilizing the strengths of MIRAGE. On the other hand, when the utilization limit is relaxed to higher values (e.g., 0.66 or 1.0), our grouping algorithm is able to consolidate larger batches of tasks into the same group, yielding greater reductions in overall processor utilization and improved scheduling efficiency.

D. MILP Efficiency

To assess the practical viability of our MILP-based batching technique for real-time systems, we conducted 600 independent optimization tests, each representing a distinct task graph. The majority of instances reach optimality under 20 seconds, with suboptimal but feasible solutions appearing even earlier. Notably, even in the rare cases where the MILP solver does not reach optimality within the time budget, the intermediate solutions offer meaningful grouping structures that respect utilization bounds and avoid cyclic dependencies. This result empirically demonstrates that MILP, when carefully constrained and guided, is well-suited for adaptive batching and scheduling of sporadic tasks in real-time systems.

Interestingly, several graphs failed to complete optimization within the one-hour time limit, as indicated by a final time near 3600 seconds. However, our analysis reveals that these same graphs often identify high-quality, sub-optimal solutions within the initial few minutes of execution, as is shown in Fig. 14. These findings suggest that even when full optimality cannot be guaranteed under strict real-time constraints, the solver is capable of producing practically useful solutions quickly.

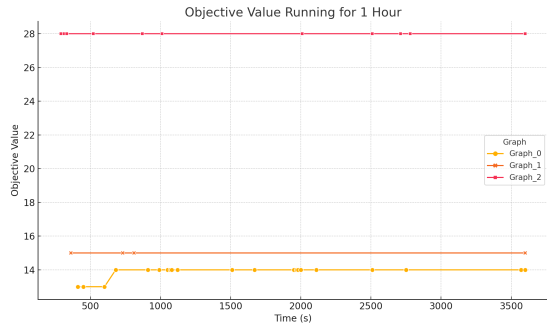

Fig. 14: Objective Value Running For 1 Hour

V. Related Work

Our work builds on prior research in real-time scheduling, task grouping, and high-frequency signal processing. Classic scheduling approaches [4], [8] and Global EDF (G-EDF) [9] are effective for moderate sampling rates, but their assumptions fail when high-frequency pipelines suffer from dominant invocation overheads.

Task grouping and node consolidation have appeared in several domains. In the automotive context, Faragardi et al. [10] merged AUTOSAR runnables to reduce inter-runnable communication costs. Amert et al. [11] demonstrated that restructuring execution units can improve temporal isolation in multicore+accelerator platforms. Voronov's recent thesis [12] examined legal conditions for merging nodes in real-time graph workloads, focusing on feasibility preservation rather than optimization. While conceptually related, these works do not address optimization-driven grouping under utilization bounds or rate constraints.

Beyond real-time systems, task grouping has been explored for overhead reduction: static methods for embedded controllers [13], dynamic clustering for streaming workloads [14], and graph partitioning methods used in machine-learning pipelines [15]. However, these approaches either overlook real-time constraints or relax the strict precedence semantics inherent to SDR pipelines.

In high-frequency signal processing, prior work has addressed jitter and latency [16], [17], largely through hardware or pipeline-level optimizations. Our MILP-based grouping is distinct in jointly enforcing (1) schedulability via utilization bounds, (2) precedence through acyclicity, and (3) communication locality—dimensions previously considered only in isolation [18], [19]. In contrast to heuristic or hardware-centric solutions, MIRAGE provides a principled optimization framework that integrates directly with G-EDF schedulability analysis, bridging system-level scheduling theory and practical high-rate SDR execution.

VI. Conclusion

Software-defined radio (SDR) has become a compelling platform for experimenting with emerging communication and signal-processing techniques, but its implementation on multicore processors continues to suffer from efficiency and predictability challenges. In this paper, we introduced MIRAGE, a MILP-based framework that systematically addresses these challenges through task grouping and inter-block merging. By formulating grouping as an optimization problem rather than relying on heuristics, MIRAGE reduces invocation overheads, improves cache locality, and provides analytically grounded schedulability guarantees under G-EDF. Our evaluation on synthetic task sets shows that this approach achieves significant utilization and latency improvements while remaining computationally tractable.

More broadly, this work highlights the potential of optimization-driven design-space exploration for real-time and embedded systems. The fundamental design constraints—timing, resource limits, and quality of service—are longstanding, but can be effectively cast into optimization models solvable with modern tools such as Gurobi. The main challenges ahead lie in extending our approach to GPU-based platforms, where massively parallel execution introduces new scheduling tradeoffs, and in developing a deeper understanding of how different workload characteristics affect cost models and solver behavior. As the community explores new paradigms, including the integration of machine learning with classical optimization, systematic frameworks like MIRAGE demonstrate that principled optimization remains a powerful and reliable foundation for advancing predictable, high-performance real-time systems.

Acknowledgments

This work was supported by DARPA's Processor Reconfiguration for Wideband Spectrum Sensing (PROWESS) program under contract HR00112490302. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] A. Eisenklam, W. Hedgecock, and B. C. Ward, "Job-level batching for software-defined radio on multi-core," in *Proc. IEEE Real-Time Syst. Symp.*, 2024, pp. 1–12.

[2] S. Goddard, "On the management of latency in the synthesis of real-time signal processing systems from processing graphs," Ph.D. dissertation, University of North Carolina, Chapel Hill, 1998.

[3] C. Liu and J. Anderson, "Supporting soft real-time dag-based systems on multiprocessors with no utilization loss," in *Proc. IEEE Real-Time Syst. Symp.*, 2010, pp. 95–106.

[4] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[5] U. Devi, "Soft real-time scheduling on multiprocessors," Ph.D. dissertation, University of North Carolina, Chapel Hill, 2006.

[6] Y. Xu, T. He, R. Sun, Y. Ma, Y. Jin, and A. Zou, "Shape: Scheduling of fixed-priority tasks on heterogeneous architectures with multiple cpus and many pes," in *Proc. IEEE/ACM Int. Conf. Computer-Aided Design (ICCAD)*, 2022, pp. 110:1–110:9.

[7] U. Devi and J. Anderson, "Tardiness bounds for global edf scheduling on a multiprocessor," in *Proc. IEEE Real-Time Syst. Symp.*, 2005, pp. 330–341.

[8] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proc. IEEE Real-Time Syst. Symp.*, 1998, pp. 4–13.

[9] S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[10] H. R. Faragardi, B. Lisper, K. Sandström, and T. Nolte, "An efficient scheduling of autosar runnables to minimize communication cost in multi-core systems," 12 2014, pp. 41–48.
[11] T. Amert, Z. Tong, S. Voronov, J. Bakita, F. D. Smith, and J. H. Anderson, "Timewall: Enabling time partitioning for real-time multicore+accelerator platforms," in *2021 IEEE Real-Time Systems Symposium (RTSS)*, 2021, pp. 455–468.
[12] S. Voronov, "Scheduling real-time graph-based workloads," Ph.D. dissertation, University of North Carolina at Chapel Hill, 2023. [Online]. Available: https://cdr.lib.unc.edu/concern/dissertations/41687t86p
[13] Y. Liu, J. Chen, and X. Hu, "Task grouping for overhead reduction in embedded systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 28, no. 7, pp. 1091–1100, 2009.
[14] J. Park and S. Kim, "Dynamic task clustering for streaming applications," in *Proc. ACM SIGPLAN Conf. Lang., Compil., Tools Embedded Syst.*, 2012, pp. 127–136.
[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2015, pp. 161–170.
[16] M. Chabukswar, D. Shasha, and M. Demirbas, "Latency and jitter minimization in high-frequency trading systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 5, pp. 1153–1166, 2018.
[17] A. Ghosh and R. Rajkumar, "Scheduling real-time gpu tasks in high-frequency trading systems," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2019, pp. 1–12.
[18] K. Lakshmanan, D. Faggioli, and G. Lipari, "Coordinated task scheduling for distributed real-time systems," *IEEE Trans. Comput.*, vol. 64, no. 5, pp. 1246–1259, 2015.
[19] P. Huang, A. Easwaran, and B. Brandenburg, "Communication-aware scheduling for real-time parallel applications," in *Proc. IEEE Real-Time Syst. Symp.*, 2016, pp. 291–302.