Optimizing Latency and Throughput of Application Workflows on Clusters

N. Vydyanathan, U. Catalyurek, T. Kurc, P. Sadayappan and J. Saltz

Technical Report
OSU-CISRC-4/08-TR17

Optimizing Latency and Throughput of Application Workflows on Clusters.[*]

N. Vydyanathan[†], U. Catalyurek[‡], T. Kurc[‡], P. Sadayappan[†], J. Saltz[‡]
[†] Dept. of Computer Science and Engineering, [‡] Dept. of Biomedical Informatics
The Ohio State University

Abstract

Scheduling, in many application domains, involves the optimization of multiple performance metrics. For example, application workflows with real-time constraints have strict throughput requirements and also desire a low latency or response time. In this paper, we present a novel algorithm for the multi-objective scheduling of workflows that act on a stream of input data. Our algorithm focuses on the two performance metrics: latency and throughput, and minimizes the latency of workflows while satisfying strict throughput requirements. We also describe steps to use the above approach to solve the problem of meeting latency requirements while maximizing throughput. We leverage pipelined, task and data parallelism in a coordinated manner to meet these objectives and investigate the benefit of task duplication in alleviating communication overheads in the pipelined schedule for different workflow characteristics. The proposed algorithm is designed for a realistic k-port communication model, where each processor can simultaneously communicate with at most k distinct processors. Evaluation using synthetic benchmarks as well as those derived from real applications shows that our algorithm consistently produces lower-latency schedules that meets throughput requirements, even when previously proposed schemes fail.

1 Introduction

Data analysis steps in a wide range of applications [14, 22, 34] can be expressed as workflows that operate on a stream of input data. Efficient scheduling of these workflows often involves optimizing several performance metrics. Scheduling with the goal of optimizing more than one performance criteria is termed as multi-objective scheduling. In this paper, we present approaches for the multi-objective scheduling of data analysis workflows that analyze a stream of data, where tasks in the workflow repeatedly receive input data items from their predecessors, compute on them, and write the output to their successors.

Efficient execution of these workflows is gauged by two metrics: latency and throughput. Latency is the time to process an individual data item through all the components of the workflow, while throughput

[*]This research was supported in part by the National Science Foundation under Grants #CCF-0342615 and #CNS-0403342.

is a measure of the aggregate rate of processing of data. It is often desirable or necessary to meet a user-defined requirement in one metric, while achieving higher performance value in the other metric and minimizing resource usage. Applications with real-time constraints, for example, can have strict throughput requirements and desire low latency, whereas interactive query processing may have strict latency constraints and desire a high aggregate query processing rate. To be able to meet requirements and minimize resource usage is also important especially in settings such as Supercomputer centers where resources (e.g., a compute cluster) have an associated cost and are contended for by multiple clients.

This paper presents a novel approach for the scheduling streaming workflows on a cluster machine such that the resulting schedule meets throughput and latency requirements. In multi-objective scheduling with two performance criteria or metrics, one metric is optimized, keeping the other a constraint. Our algorithm optimizes latency of streaming workflows while meeting throughput requirements. We also describe steps to use the above approach to solve the problem of meeting latency requirements while maximizing throughput.

Our algorithm employs pipelined-, task- and data-parallelism in an integrated manner to meet the above described performance goals. The algorithm is designed to satisfy the throughput requirements by leveraging pipelined parallelism and through intelligent clustering, duplication and/or replication of tasks. *Pipelined-parallelism* is the concurrent execution of dependent tasks that operate on different instances of the input data stream, while *data-parallelism* is the concurrent processing of multiple data items by replicas of a task. Latency is minimized by exploiting *task-parallelism*, which is the concurrent execution of independent tasks on the same instance of the data stream, and minimizing communication costs along the critical path of the task graph through duplication and clustering. We employ a flexible k-port communication model, where each processor can communicate with at most k distinct processors simultaneously. The value of k is determined by the ratio of the network-card capacity on a processor and the link capacity.

We compare the proposed approach against two previously proposed schemes: Filter Copy Pipeline (FCP) [28] and EXPERT (EXploiting Pipeline Execution undeR Time constraints) [18]. Evaluations are done using synthetic benchmarks and task graphs derived from real applications in the domains of Image Analysis, Video Processing and Computer Vision [2, 18, 27, 1]. We show that our algorithm is able to generate low latency schedules that meet throughput requirements, even when previously proposed approaches fail.

2 Related Work

Several algorithms for scheduling streaming workflows focus on maximizing the throughput. These algorithms leverage pipelined-parallelism between dependent tasks to improve the throughput. Lee et al. [24] propose a three-step mapping methodology for maximizing the throughput of applications comprising of a sequence of computation stages, each consisting of a set of identical sequential tasks. Jonsson et al. [20] and Hary and Ozguner [19] discuss heuristics for maximizing the throughput of directed acyclic task graphs on multiprocessor systems using point-to-point networks, while Yang [35] has presented an approach for resource optimization under throughput constraints. Benoit and Robert [8] have addressed the problem of mapping pipeline skeletons of linear chains of tasks on heterogeneous systems and Suhendra et al. [30] have proposed an integrated approach for task scheduling and scratch-pad memory allocation based on integer linear programming for multiprocessor system-on-chip architectures. Agarwalla et al. [2, 3] propose an adaptive scheduler for maximizing throughput of streaming applications on the Grid. All of these techniques optimize the throughput metric under the assumption that tasks cannot be replicated. Another work by Chen et al. [12, 13] presents a resource allocation algorithm based on minimum spanning trees, that maps the stages of a communication intensive distributed data stream processing application to processing nodes along high bandwith paths in a Grid environment. This method, however assumes that the computation is not a bottleneck and that the data arrival rates significantly decrease as they pass through multiple processing stages, which is true for applications that filter data while processing.

Though many papers focus on optimizing latency or throughput in isolation, very few address both. Subhlok and Vondran [29] have proposed a dynamic programming solution for optimizing latency under throughput constraints for applications composed of a chain of data-parallel tasks. Choudhary et al. [15] address the optimal processor assignment for pipelined computations of non-replicable parallel tasks with series-parallel dependences, that minimizes latency under throughput constraints. Benoit and Robert [9] study the theoretical complexity of latency and throughput optimization of pipeline and fork graphs with replication and data-parallelism under the assumptions of linear clustering and round-robin processing of input data items. They also propose heuristics for optimizing latency and throughput of linear chain workflows without replication [7, 5]. They also address optimizing latency and reliability (through redundancy) of linear chain workflows [6]. The above algorithms are however, specific to certain task graph topologies.

Spencer et al. [28] present the Filter Copy Pipeline (FCP) scheduling algorithm for optimizing latency and throughput of arbitrary application DAGs on heterogeneous resources. FCP computes the number of copies of each task that is necessary to meet the aggregate production rate of its predecessors and maps the copies to processors that yield their least completion time. Another closely related work is by Guirado et al. [18], which proposes a task mapping algorithm called EXPERT (EXploiting Pipeline Execution undeR Time constraints) that minimizes latency of streaming applications, while satisfying a given throughput constraint. EXPERT identifies maximal clusters of tasks that can form synchronous stages that meet the throughput constraint and maps tasks in each cluster to the same processor so as to reduce communication overheads and minimize latency. Our proposed algorithms aims to generate lower latency schedules that meet the throughput requirement and use fewer resources than the schedules generated by FCP and EXPERT.

3 Task Graph and System Model

An application workflow can be represented as a connected, weighted directed acyclic graph (DAG), $G = (V, E)$, where V, the set of vertices, represents non-homogeneous sequential application components (tasks) and E, the set of edges, represents precedence constraints and data dependences. There are two distinguished vertices (tasks) in the task graph: the *source task* which precedes all other tasks and the *sink task* which succeeds all other tasks.

The task graph G acts on a stream of data, where each task in G repeatedly receives input data items from its predecessors, computes on them, and writes the output to its successors. If G acts on a stream of independent data items, i.e., the processing of each data item through the workflow is independent of the processing of other data items, multiple data items can be processed concurrently by replicas of tasks.

The weight of a vertex (task), $t_i \in V$, is its execution time to process a single data item, $et(t_i)$. The weight of an edge $e_{i,j} \in E$, $wt(e_{i,j})$ is the communication cost measured as the time taken to transfer a single data item of size $d_{i,j}$ between t_i and t_j. $wt(e_{i,j}) = \frac{d_{i,j}}{bw_{i,j}}$, where $bw_{i,j}$ is the minimum of disk or memory bandwidth of the system depending on the location of data and the network bandwidth. The length of a path in a DAG G is the sum of the weights of the tasks and edges along that path. The *critical path* of G, denoted by $CP(G)$, is defined as the longest path in G. The *top level* of a task t in G, denoted by $topL(t)$, is defined as the length of the longest path from the source task to t, excluding the weight of t. The *bottom level* of a task t in G, denoted by $bottomL(t)$, is defined as the length of the longest path from t to the sink, including the weight of t. Any task t with maximum value of the sum of $topL(t)$ and $bottomL(t)$ belongs to a critical path in G.

The task graph is assumed to be executed on a homogeneous fully connected compute cluster, with each compute node having local disks. The scheduling algorithm assumes a k-port communication model; each processor can send data to or receive data from at most k distinct processors at the same time. The value of k is determined as the ratio of the network card capacity and the link capacity. Multiple messages between any pair of processors are transferred serially, i.e., in a time step there can be at most one message in transit on a link between a pair of processors. The system model assumes overlap of computation and communication, as most clusters today are equipped with high performance interconnects which provide asynchronous communication calls.

It is assumed that the execution behavior of the tasks in the workflow is not strongly dependent on the properties of the input data items. Hence, profiling the workflow on a few sample data items, gives a reasonable measure of the execution times of the constituent tasks. This is true for many pipelined applications like image processing [22] where there is not a significant variation in runtime/size across different images. However, in the presence of significant fluctuations, the latency and throughput of our schedules would also fluctuate. We expect the impact on latency to be more pronounced as fluctuations in runtimes and communication costs along the critical path have an additive effect on latency. In contrast, as throughput depends only on the bottleneck task or communication, there is a greater chance for masking fluctuations.

4 Multi-objective Scheduling Heuristic

Given a workflow-DAG G, P homogeneous processors and a throughput constraint T, our multi-objective scheduling heuristic (MOS) generates a mapping and schedule of G on P that minimizes the latency while satisfying the throughput constraint. MOS is designed for a k-port communication model and takes into account communication contention and its impact, if it is on the critical path, while deriving low latency schedules that meet the throughput requirement. Consider the workflow DAG in Figure 1(a). All tasks take 10 time units to process a data item and all edges have a cost of 8 time units to transfer a data item. For simplicity, assume a negligible throughput constraint. Figure 1(b) and (c) depict two possible mappings/schedules with a latency of 48 time units if we assume k to be very large. Here, $t_{i'}$ and $t_{i''}$ are duplicates of t_i, for $i = 1, 3$. Mapping B is obtained assuming a fully multi-port model, which corresponds to the algorithm presented in our previous work [33, 32] along with duplication. When k is 2, mapping A has a latency of 48 time units, while mapping B has a longer latency of 56 time units. As a processor can communicate with at most 2 distinct processors simultaneously, communications along edges $e_{4,7}$ and $e_{6,7}$ are serialized for mapping B, resulting in a longest path through $t_{1'}$, $t_{3'}$, t_4, $e_{4,7}$, $e_{6,7}$ and t_7 of length 56. Though communications are also serialized in mapping A, these do not lie on the critical path and hence latency is not increased. MOS accurately models the impact of communication

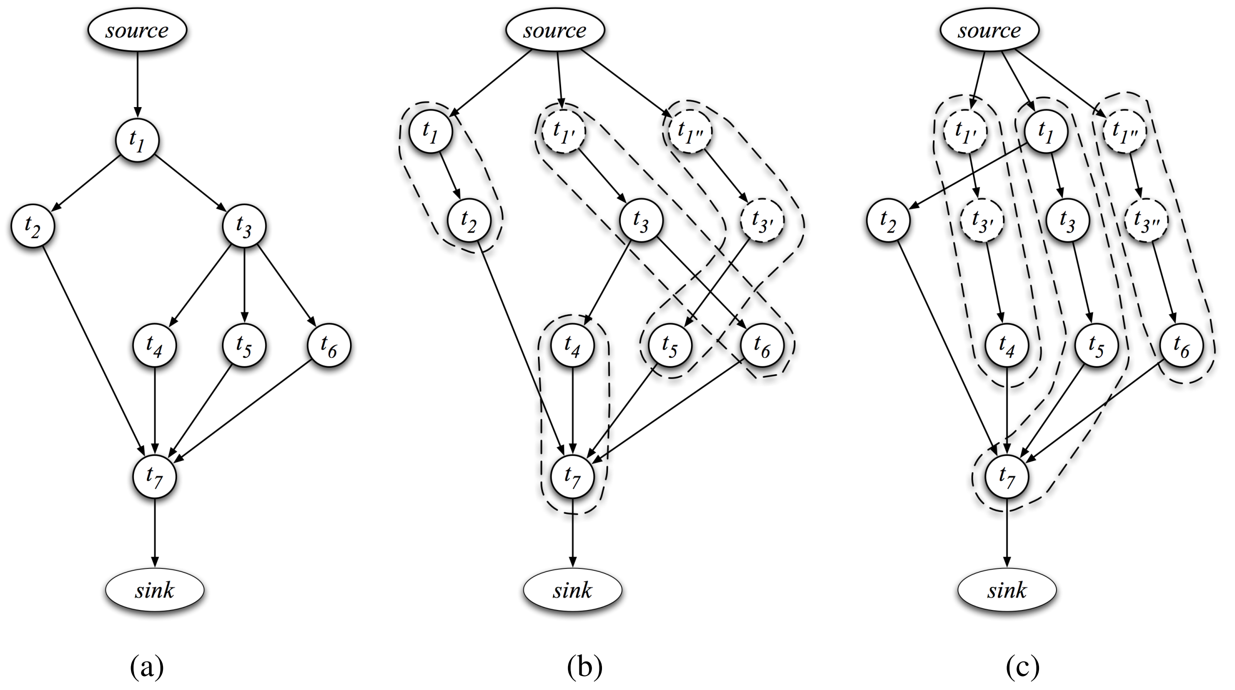


Figure 1. (a) Sample workflow DAG, (b) Mapping A and (c) Mapping B.

contention on the latency and throughput of pipelined schedules and hence derives schedules that meet the performance criteria (in this case, mapping A) in the presence of contention, while algorithms that assume a fully multi-port model fail.

MOS generates two schedules: a non-pipelined schedule and a pipelined schedule, and chooses the one that yields lower latency while meeting the throughput requirement. The non-pipelined schedule is generated using a priority-based list-scheduling heuristic [23], where the tasks are scheduled in the decreasing order of their bottom-levels, to processors that yield their least completion time. If L is the latency of this schedule, the throughput achieved is $\frac{1}{L}$.

The pipelined schedule is generated in three phases. In the first phase, a schedule that meets the throughput requirement is obtained, assuming an unbounded number of processors, using the *Satisfy Throughput Heuristic* (STH). The second phase uses the *Processor Reduction Heuristic* (PRH) to limit the number of processors used in the schedule to what is available. The third phase focuses on minimizing the latency of the pipelined schedule through the *Latency Minimization Heuristic* (LMH). Though the primary objective of the first two phases is not latency optimization, preference is given to scheduling decisions that minimize latency. The following sub-sections describe these three phases in greater detail.

4.1 Phase 1: Satisfying the Throughput Constraint

Theorem 1 *Given a workflow-DAG $G = (V, E)$ that acts on a stream of independent data items, the maximum achievable throughput T_{max}, on P homogeneous processors is given by $\frac{P}{\sum_{t \in V}(et(t))}$, where $et(t)$ is the time taken by task t to process a single data item.*

Proof The minimum amount of work to be done to process one data item through G is given by $\sum_{t \in V}(et(t))$. The minimum work to be done per unit time to process T_{max} data items is $T_{max} \times$

$\sum_{t \in V}(et(t))$. Since, we have atmost P processors in the system and all the tasks can be replicated as they process a stream of independent data items, $P = T_{max} \times \sum_{t \in V}(et(t))$, which implies that T_{max}, the maximum achievable throughput is $\frac{P}{\sum_{t \in V}(et(t))}$.

T_{max} can be achieved by grouping together all tasks in G to form a task-cluster and executing P replicas of this task-cluster, each replica mapped to a unique processor and processing a different input data item. However, this mapping suffers from a large latency as it fails to exploit parallelism between *concurrent tasks* in G. Concurrent tasks refers to independent tasks in G that can be executed concurrently. For the sake of presentation, the rest of this section assumes that G acts on a stream of independent data items and hence all tasks can be replicated. However, the heuristics described here can be applied when processing of a data item is dependent on the processing of certain other data items (i.e replication of tasks is not allowed), by enforcing the weight of every task-cluster to be $\leq \frac{1}{T}$, for a given throughput constraint $T \leq T_{max}$. T_{max} in this case, is the reciprocal of the weight of the largest task in G.

Given a throughput constraint $T \leq T_{max}$, the Satisfy Throughput Heuristic (STH) meets the throughput constraint by replicating, clustering and duplicating tasks. Algorithm 1 illustrates STH. Each task $t_i \in V$ is initially mapped to a separate task-cluster C_i. M denotes the unordered list of all the task-clusters. The time taken by a task-cluster to process one data item is given by the sum of the execution times of its constituent tasks to process a data item, i.e., $et(C_i) = \sum_{\forall t \in C_i} et(t)$. Initially, $et(C_i)$ is the same as $et(t_i)$. The number of replicas of C_i, $nr(C_i)$, required to satisfy T is computed as $nr(C_i) = T \times et(C_i)$. When there is no throughput constraint only one replica is created, $nr(C_i) = 1$.

Given the initial mapping M, the corresponding pipelined schedule is obtained by running each replica of a task-cluster in M on a unique virtual processor. Tasks within a task-cluster are executed in the decreasing order of their bottom-levels and iterate over the instances of the input stream. For example, if tasks t_1 and t_2 are mapped to task-cluster C where $bottomL(t_1) > bottomL(t_2)$, and $nr(C)$ is 2, each replica of C is mapped to a unique virtual processor and t_1 processes a data item followed by t_2 in the steady state on each processor. If the throughput of the pipelined schedule computed as described in Section 4.1.1 (step 7) is less than the required throughput, the mapping M is refined repeatedly (by replicating, duplicating or clustering) (step 9-18), until the throughput requirement is satisfied. The details of the refinement process is explained in Section 4.1.3.

4.1.1 Estimating Throughput of a Pipelined Schedule

In this section, we outline an approach to estimate the throughput of a pipelined schedule, while modeling communication contention in good detail. The throughput of a pipelined schedule involves two components: the processing or computation rate and the data transfer or communication rate. Given a mapping M comprising of a set of task-clusters, the processing or computation rate, $pr(M)$, is determined by the slowest task-cluster, i.e., $pr = \min_{\forall C_i \in M} \frac{\lceil nr(C_i) \rceil}{et(C_i)}$.

To estimate the data transfer rate, the edges in the workflow DAG G are considered as special types of tasks (communication tasks) and are scheduled under a k-port communication model as illustrated below. Figure 2(a) represents an example DAG with two types of tasks - computation tasks and communication tasks. Let each task be mapped to a separate task-cluster with one replica each, i.e let t_1, t_2, t_3 and t_4 be mapped to $P1$, $P2$, $P3$ and $P4$ respectively. Each task takes 10 time units to process a data item. Edges $e_{1,2}, e_{1,3}, e_{2,4}$ and $e_{3,4}$ have weights 8, 5, 9 and 9 respectively. In the steady state, each edge is ready to transfer a data item. To schedule these transfers under the k-port model, the communication

Algorithm 1 STH: Satisfy Throughput Heuristic

```
1:  function STH($G, T$)                    ▷ $G$: workflow DAG, $T$: throughput constraint $\leq T_{max}$
2:      $M \leftarrow < C_i \mid C_i = \{t_i\}$ for all $t_i \in V >$ ▷ $M$ is an unordered list of task-clusters, initially each task $t_i$ is
            a separate task-cluster $C_i$
3:      if $T > 0$ then
4:          For all $C_i, nr(C_i) \leftarrow T \times et(C_i)$
5:      else
6:          For all $C_i$, $nr(C_i) \leftarrow 1$
7:      $T_{curr} \leftarrow$ getThroughput($M$)
8:      if $T_{curr} < T$ then
9:          for all edges $e_{i,j}$ with $\frac{\min(nr(t_i),nr(t_j))}{wt(e_{i,j})} < T$ do
10:             Evaluate the possibility of duplication, clustering and replication and perform the one that yields
                    the least increase in latency. To break ties choose the one that uses least number of extra
                    processors.
11:             Update $M$
12:         $T_{curr} \leftarrow$ getThroughput($M$)
13:     while $T_{curr} < T$ do
14:         For all edges that are scheduled on the channel with the maximum cycle time, evaluate the possibility
                of duplication and clustering.
15:         Pick the edge that yields the least increase in latency (break tie by choosing the largest weighted one)
                with the above techniques.
16:         Evaluate the possibility of replicating edges to meet the throughput constraint.
17:         Perform duplication, clustering or replication, whichever yields the least increase in latency.
18:         Update $M$
19:         $T_{curr} \leftarrow$ getThroughput($M$)
20:     return $M$
```

tasks are prioritized in the decreasing order of their bottom-levels and scheduled in priority order to the k communication channels in the earliest available idle slot. Such a schedule takes into account the contention between communications scheduled on the same channel between overlapping groups of processors. Figure 2(b) illustrates the communication schedule assuming a one-port model, i.e., k = 1. This schedule is repeated periodically and each schedule instance is denoted as a cycle. The instance of data item transferred along an edge for a given cycle is also shown. For example, in cycle n, edges $e_{1,2}$ and $e_{1,3}$ transfer the n-th item while, edges $e_{2,4}$ and $e_{3,4}$ transfer the $(n-1)$-th item.

Once the communications are scheduled, the communication throughput is computed as follows. For every edge $e_{i,j}$ the number of parallel transfers $pt(e_{i,j})$ is computed as $\min(nr(t_i), nr(t_j))$. In the example, as each task has one replica, for all edges $e_{i,j}$, $pt(e_{i,j})$ is 1. On every communication channel k_i, the minimum time after which the periodic schedule can repeat (also called minimum cycle time), $minCT(k_i)$ can be estimated as the difference between the completion time of the last transfer and the start time of the first transfer in a cycle. In the example, if we term the communication channel on processor P_1 as k_1 and so on, $minCT(k_1)$ is 13, $minCT(k_2)$ is 17, $minCT(k_3)$ and $minCT(k_4)$ is 18. However, as every communication involves a pair of processors, it may not be possible to start the transfers for the next cycle after the minimum cycle time due to contention. In the worst case, this periodic schedule can repeat after all the contending transfers for the current cycle are completed. To compute the set of contending transfers, we construct an undirected graph (called the contention

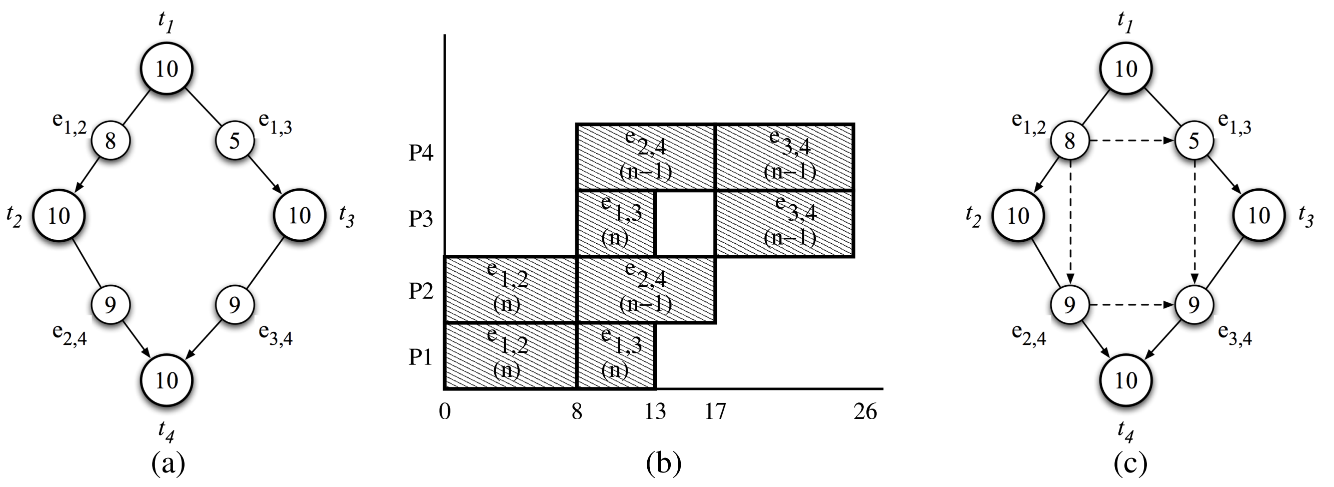


Figure 2. (a) Workflow DAG G, (b) Communication schedule on a 1-port system and (c) Schedule DAG G'.

graph, say G_c) with processors as vertices and edges between pairs of processors that are involved in a data transfer. The connected components of G_c identifies the group of processors involved in contending transfers. For each of these connected components, (let $Conn(G_c)$ denote a connected component in G_c), we compute the time taken to complete the corresponding set of data transfers as the maximum of the minimum cycle times of the communication channels of the processors in the connected component, i.e $\max_{\forall k_i \in Conn(G_c)} minCT(k_i)$. In the example, all four processors belong to one connected component. Hence the time taken to complete the corresponding transfers for one cycle is $\max(minCT(k_1), minCT(k_2), minCT(k_3), minCT(k_4))$ which is 18. The data transfer rate for a connected component, $Conn(G_c)$ is given by
$\min_{\forall k_i \in Conn(G_c)} \frac{\min_{\forall e_{i,j} \in k_i} pt(e_{i,j})}{\max_{\forall k \in Conn(G_c)} minCT(k)}$, which is $\frac{1}{18}$ for the given example. The overall data transfer rate $dr(M)$ is the minimum of the data transfer rates of the connected components of G_c. As we assume that computations and communications can overlap, the overall throughput of the mapping M is given by $\min(pr(M), dr(M))$. Please note that edges within a task-cluster incur zero transfer costs and hence are not scheduled.

4.1.2 Estimating Latency of a Pipelined Schedule

Given a mapping M, the latency of the corresponding pipelined schedule is estimated by the critical path length of the DAG G' that represents all dependences in the schedule. To generate G' from G, for every pair of concurrent tasks that are mapped to the same task-cluster, a zero weight pseudo-edge originating from the task with the larger bottom level is added in G'. Communication edges between tasks mapped to the same task-cluster have zero weight in G'. In addition, zero weight pseudo-edges are added between communication edges that have a dependency in the communication schedule under the k-port model. For the schedule in Figure 2(b), the schedule DAG G' is given in Figure 2(c). The latency of the pipelined schedule, which is given by the length of the critical path $(t_1, e_{1,2}, t_2, e_{2,4}, e_{3,4}, t_4)$, is 56 time units.

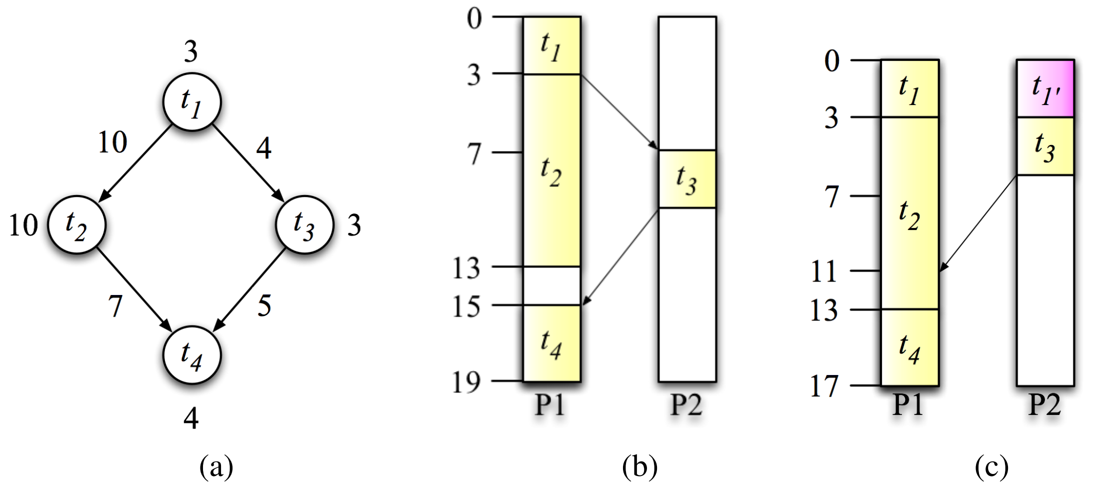


Figure 3. (a) Sample application DAG and Schedule (b) without duplication (c) with duplication.

4.1.3 Refining Task Mapping to Improve Throughput

The initial mapping of each task to a separate task-cluster is repeatedly refined by STH until the throughput requirement is met. As As the number of replicas required is computed based on the throughput constraint, the bottleneck is the data transfer rate, i.e the communication throughput. To improve the communication throughput, our scheduling heuristic alleviates communication overheads by techniques like task duplication and clustering. In addition, the communication throughput can be increased by improving the degree of parallel transfers by adding replicas to tasks. Each of these three techniques: duplication, clustering and replication, is described below.

Task duplication is a technique used in DAG scheduling to alleviate communication costs through the redundant allocation of tasks to multiple processors [21, 25, 17, 16, 26, 4, 11, 10]. Figure 3 illustrates how duplication can be beneficial in minimizing the parallel completion time (latency). The execution times of the tasks in the example is given by the vertex weights and the communication costs are denoted by the edge weights (Figure 3(a)). The DAG is assumed to be scheduled on 2 homogeneous processors. A scheduling algorithm without task duplication can achieve a minimum latency of 19 (Figure 3(b)). A duplication-based scheme can execute a redundant copy of task t_1 on processor $P2$, and thereby avoid communication between tasks t_1 and t_3. This results in a lower latency of 17 (Figure 3(c)).

Duplication involves broadcasting input data from predecessors to all the duplicates of a task. If any these extra communications violates the throughput requirement, STH investigates the possibility of duplicating the parent task and its ancestors. As duplication involves redundant processing, extra processors may be required to meet the throughput requirement. STH performs task duplication, only if it does not require more processors than available and does not involve expensive communications that violate the throughput constraint. Due to the extra communications involved in broadcasting input data to task duplicates, duplication may increase the latency.

Clustering is a technique to avoid expensive communications by mapping the communicating tasks to the same processor (task-cluster). Clustering does not require extra processors but can increase the latency. In workflow DAGs, where a parent task has multiple children and communicates large amounts of data to each of them, avoiding the communications by clustering can neglect the task-parallelism in the DAG, leading to large latencies.

Replication can be used to improve communication throughput by increasing the degree of parallel transfers by adding replicas to the tasks involved in the bottleneck communications. For example, consider the transfer of data between tasks t_1 and t_2, whose execution times to process a single data item is 10 units of time each. Let the time taken to transfer a data item between these tasks be 50 time units over a free channel. In a pipelined execution of t_1 and t_2 on two processors, the throughput, which is limited by the data transfer rate, is 1 data item processed every 50 units of time. If we had 2 additional processors, we could replicate t_1 and t_2 to one extra processor each.This would double the aggregate transfer rate, if there were distinct paths between every pair of processors in the system. Please note that replication does not affect the workflow latency and is different from duplication. In duplication, redundant copies of a task are allocated to multiple processors, where each copy does the same computation on the same data item. In contrast, replicas of a task process different data items. STH replicates tasks only if there are enough available processors.

4.2 Phase 2: Adjusting Number of Processors

Once a mapping that meets the throughput constraint is obtained in Phase 1, Phase 2 limits the number of processors used to what is available. The total number of processors required to execute $nr(C_i)$ copies of each task-cluster C_i, where each copy is mapped to a unique processor, is $P' = \sum_{C_i \in M} \lceil nr(C_i) \rceil$. If $P' > P$, we use the Processor Reduction Heuristic, PRH, described in this section, to recursively merge certain task-clusters to obtain a schedule that uses as many processors as available.

Proposition 1 *If task-clusters C_i and C_j are merged and P_i and P_j are the number of processors required to run the replicas of C_i and C_j respectively, i.e $P_i = \lceil nr(C_i) \rceil$ and $P_j = \lceil nr(C_j) \rceil$, the number of processors required to run the replicas of the new task-cluster formed that meets the throughput constraint is either $P_i + P_j$ or $P_i + P_j - 1$.*

Definition 1 *Task-clusters C_i and C_j are "connected" if there exists some task t_a in C_i and some task t_b in C_j such that $e_{a,b}$ is an edge in G.*

Definition 2 *Task-clusters C_i and C_j are "not concurrent" if for all pairs of tasks (t_a, t_b), $t_a \in C_i$ and $t_b \in C_j$, t_a is not concurrent to t_b in G.*

Definition 3 *Resource wastage of a task-cluster C is defined as $\lceil nr(C) \rceil - nr(C)$.*

The pseudo code of PRH is illustrated in Algorithm 2. Step 4 of the algorithm considers all pairs of task-clusters that when merged would reduce the number of processors used by 1. Among these, PRH picks the task-cluster pair that yields the largest decrease in latency when merged. To break ties, preference is given to task-clusters that are connected, not concurrent, and which produce the largest resource wastage, in that order (step 6). Merging connected task-clusters helps in avoiding some of the communication costs and merging task-clusters that are not concurrent avoids serializing concurrent tasks in G. Giving preference to task-cluster pairs that yield a larger resource wastage reduces the possibility of fragmentation. Steps 5-9 are repeated as long as there are task-cluster pairs that reduce the processor count and $P' > P$. After all possible task clusterings, if the resource usage is still greater than P at step 10, defragmentation is done in steps 11-12 where the task-clusters that produce the largest resource wastage are merged. To break ties, the one that causes the largest decrease in latency is chosen. The outer-loop (steps 3-13) are repeated until the resource usage is lesser than or equal to P. At the end of the processor reduction phase, a mapping M is obtained that meets the throughput constraint and uses $\leq P$ processors.

Algorithm 2 PRH: Processor Reduction Heuristic

```
1: function PRH($M$)                                  ▷ $M \leftarrow$ mapping returned by STH
2:     $P' = \sum_{C_i \in M}(\lceil nr(C_i) \rceil)$
3:     repeat
4:         $\mathcal{C}' \leftarrow \{(C_i, C_j) \mid C_i \in M \wedge C_j \in M \wedge \lceil nr(C_i) + nr(C_j) \rceil < (\lceil nr(C_i) \rceil + \lceil nr(C_j) \rceil)\}$
5:         while $\mathcal{C}'$ not empty $\wedge (P' > P)$ do
6:             Pick the task-cluster pair $(C_i, C_j)$ from $\mathcal{C}'$ that yields the largest decrease in latency when merged.
                   Preference is given to task-clusters that are connected, not concurrent and which produce the
                   largest resource wastage when merged.
7:             Merge $C_i$ and $C_j$ and update $M$
8:             $P' \leftarrow P' - 1$
9:             Update $\mathcal{C}'$ the task with the larger bottom-level.
10:        if $P' > P$ then
11:            Pick the task-cluster pair $(C_i, C_j)$ that yields the maximum value of $\lceil (nr(C_i) + nr(C_j)) \rceil -$
                   $(nr(C_i) + nr(C_j))$ and the largest decrease in latency when $C_i$ and $C_j$ are merged.
12:            Merge $C_i$ and $C_j$ and update $M$
13:    until $P' \leq P$
14:    return $M$
```

4.3 Phase 3: Minimizing Latency

The Latency Minimization Heuristic (LMH) is called to refine the mapping obtained by PRH to further optimize the latency. Given a mapping M that meets the throughput constraint and uses fewer than or equal to P processors, LMH minimizes latency by reducing communication overheads along the critical path.

The task-clusters in M are considered by LMH as indivisible macro-tasks. A macro-task therefore, may contain one or more tasks. The incoming and outgoing edges of a macro-task is the union of the incoming and outgoing edges, respectively, of the tasks that it contains, without considering edges between tasks belonging to the macro-task. Hence, the term task in Theorem 2 is the same as macro-task in the case where multiple tasks are mapped to same task-cluster by PRH.

Theorem 2 *Let G' and M denote a schedule and mapping of G that meets the throughput constraint and uses $\leq P$ processors (For a given mapping M, please note that G' , the schedule DAG, is generated from G through the steps described in Section 4.1.2). Let $e_{i,j}$ be an edge in G' from task/macro-task t_i to t_j such that the in-degree(t_i) = in-degree$(t_j) = 1$ and the out-degree(t_i) = out-degree$(t_j) = 1$ (i.e. t_i and t_j are connected along a linear chain in that order). Let t_k be the parent of t_i and t_l be the child of t_j. If $wt(e_{i,j}) > wt(e_{k,i}) + wt(e_{j,l})$, it is optimal to merge t_i and t_j to a single task-cluster, assuming that all tasks can be replicated. If replication is not allowed, t_i and t_j can be merged to a single task-cluster only if $et(t_i) + et(t_j) \leq \frac{1}{T}$ and $e_{i,j}$ satisfies the above condition.*

Proof Let us assume that in the optimal mapping M_{opt} that minimizes the latency while meeting the throughput constraint, t_i and t_j are mapped to different task-clusters. Consider an alternative mapping M', where t_i and t_j are pulled out from their respective task-clusters and mapped to a new one. Replication ensures that the mapping M' meets the throughput constraint. For the case when replication is not allowed, as $et(t_i)+et(t_j) \leq \frac{1}{T}$, throughput constraint is satisfied. As the replicas of each task/macro-task

Algorithm 3 LMH: Latency Minimization Heuristic

```
1: function LMH($M$)                                   ▷ $M \leftarrow$ mapping returned by PRH.
2:     repeat
3:         $< G', L > \leftarrow$ getLatency($M$)   ▷ $G'$ is DAG with all dependences in the pipelined schedule, and $L$ is
               latency
4:         Collapse edges in $G'$ that satisfy Theorem 2 by merging the incident task-clusters
5:         For every edge $e_{i,j}$ in $CP(G')$, evaluate the decrease in latency due to clustering and duplication
6:         Pick edge $e_{i,j}$ in $CP(G')$ that does not increase the latency and has maximum value of
               $\min(wt(e_{i,j}), CPL(G') - LBL(G))$   ▷ $CPL(G')$ is Critical Path Length of $G'$, $LBL(G)$ is
               Lower Bound on Latency of $G$.
7:         Perform clustering or duplication, whichever yields the maximum decrease in latency
8:         Update $M$
9:         $< G', L > \leftarrow$ getLatency($M$)
10:    until For all edges $e_{i,j}$ in $CP(G')$, both duplication and clustering cause an increase in latency
11:    return $M, L$
```

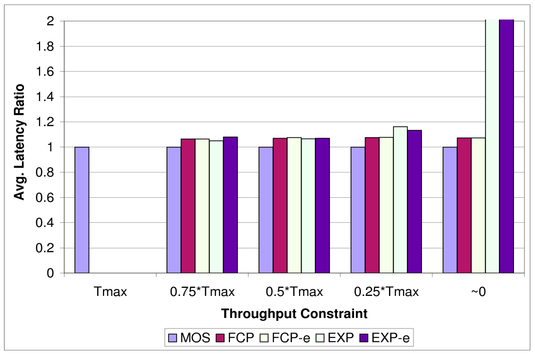
in M, the mapping generated by PRH (the number of replicas of a task/macro-task is 1 when replication is not allowed) could be run on disjoint subsets of processors using $\leq P$ processors, by applying theorem 1, any clustering among the tasks/macro-tasks in M will use $\leq P$ processors and will meet the throughput constraint. Therefore, M' uses $\leq P$ processors and meets the throughput constraint. As t_i and t_j are mapped to the same task-cluster in M', the length of the longest path through them is reduced by atleast $wt(e_{i,j}) - (wt(e_{k,i}) + wt(e_{j,l}))$. Thus, M' always yields lower latency (if $e_{i,j}$ lies on the critical path) or same latency (if $e_{i,j}$ does not lie on the critical path) as M_{opt}, which contradicts our assumption that M_{opt} was optimal. Hence, merging t_i and t_j to a single task-cluster is optimal.

Definition 4 *The lower bound on the latency of a DAG G is the length of the critical path in G, assuming all edges have zero weights.*

Algorithm 3 describes LMH. In steps 3-6, the DAG G', representing the dependences in the pipelined schedule is constructed and for all edges along $CP(G')$, the decrease in latency due to duplication and clustering is evaluated. Among edges in $CP(G')$, LMH picks the edge with the largest maximum possible decrease in latency. For this edge, clustering or duplication, whichever yields the maximum decrease in latency is performed and the mapping M is accordingly updated. The outer-loop of steps 2-10 is repeated until for all edges in $CP(G')$, both clustering and duplication increase the latency.

5 Performance Analysis

This section evaluates the performance of our multi-objective scheduling heuristic (MOS) against previously proposed schemes: Filter Copy Pipeline (FCP) [28] and EXploiting Pipeline Execution undeR Time constraints (EXPERT, abbreviated as EXP in this section) [18], and FCP-e and EXP-e, modified versions of the above schemes. When FCP and EXP fail to utilize all processors and do not meet the throughput requirement T, FCP-e recursively calls FCP on the remaining processors until T is satisfied or all processors are used, while EXP-e replicates the task-clusters by dividing the remaining processors among them in the ratio of their weights. Evaluations are done using both synthetic task graphs and those derived from real applications, using simulations.



(a)

T	MOS	FCP	FCP-e	EXP	EXP-e
T_{max}	1	-	-	-	-
0.75*T_{max}	0.96	0.97	0.97	1	1
0.5*T_{max}	0.82	0.89	0.89	0.72	0.94
0.25*T_{max}	0.70	0.79	0.79	0.42	0.98
≈ 0	0.36	0.71	0.71	0.03	1

(b)

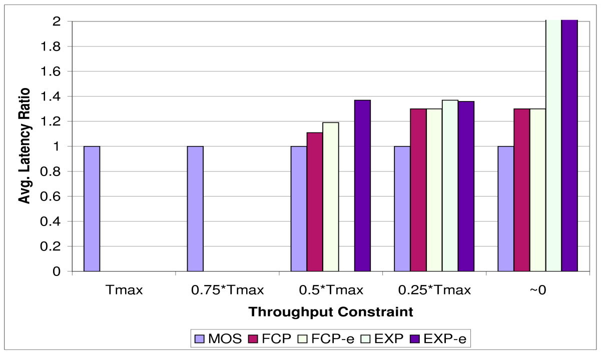
Figure 4. Performance on synthetic graphs on 32 processors for k=2, CCR=0.1 (a) Avg. Latency Ratio and (b) Avg. Utilization Ratio. (The missing values indicate that the corresponding algorithm could not meet the throughput requirement.)

5.1 Synthetic Task Graphs

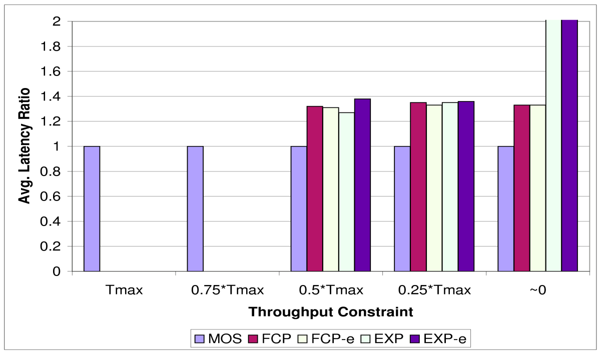
For applications that are linear chains of tasks, MOS generates schedules with optimal latency and throughput by replicating the chain of tasks on every processor. However, on arbitrary DAGs, the behavior is non-trivial. To study this, a set of 30 synthetic graphs were generated using a DAG generation tool [31], with number of tasks per task graph varying from 10 to 50. The average out-degree and in-degree per task was 4. The computation time of each task was generated as a uniform random variable with mean equal to 30. The communication to computation ratio (CCR) was varied as 0.1, 1 and 10 and the communication cost of an edge was randomly selected from a uniform distribution with mean equal to 30 (the average computation time) times the specified value of CCR. The value of k was varied as 2, 4 and 8.

Figure 4 shows the relative performance of the algorithms on 32 processors when k is 2 and CCR is 0.1. Figure 4(a) plots the average latency ratio. Latency ratio is the ratio of the latency of the schedule generated by an algorithm to that of MOS. The x-axis is the throughput constraint, which is decreased from the maximum achievable throughput (T_{max}) in steps of 0.25. ≈ 0 refers to the case when there is no throughput constraint (or negligibly small). The missing bars in the graph indicates that the corresponding algorithm could not meet the throughput requirement. Figure 4(b) shows the average utilization ratio. The utilization ratio is given by the ratio of the number of processors used by an algorithm to the total number of available processors. Though FCP and EXP assume a fully multi-port communication model, i.e., $k = \infty$, for EXP, we estimate the latency and throughput of its schedules under the constraint of a k-port model. However, as FCP independently assigns replicas of a task to different processors and does not form task-clusters, estimating the latency and throughput of its schedules under a k-port model becomes complex. Hence, for FCP, we estimate the latency and throughput assuming a fully multi-port model. Therefore, the actual benefit seen with respect to FCP and FCP-e on a k-port model could be more than what is reported in this section.

We find that MOS is consistently able to generate schedules that meet the throughput constraint, while FCP and EXP fail at large throughput requirements. Though FCP replicates tasks, it computes the number of replicas independent of the number of processors and fails to refine the number of replicas when it maps multiple tasks to the same processor. EXP does not replicate tasks. The modified versions



(a)



(b)

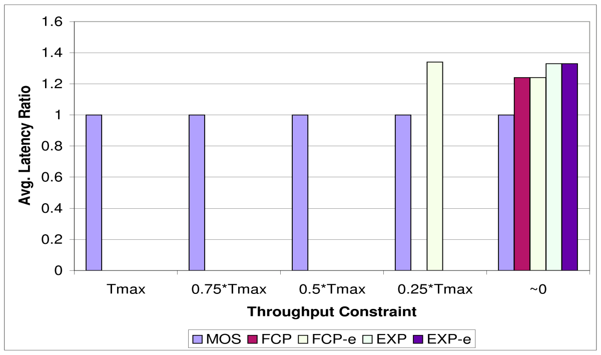
Figure 5. Performance on synthetic graphs on 32 processors for CCR=1 (a)k=4 (b)k=8. (The missing bars indicate that the corresponding algorithm could not meet the throughput requirement.)

are designed to overcome some of these limitations and hence, meet the constraint in some of the cases where FCP or EXP fail as seen in some following results.

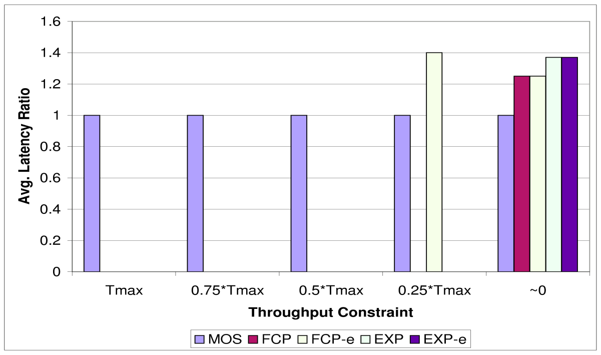
We also note that MOS generates lower latency schedules than the other schemes. FCP generates up to 7% longer latencies than MOS, while EXP generates up to 16% longer latencies when throughput constraint is 0.25*T_{max}. As EXP creates maximal task-clusters with weights $\leq \frac{1}{T}$, for negligible throughput constraint, it groups all tasks to a single task-cluster and hence generates large latency schedules. For FCP-e, we used the smallest of the latencies of all the workflow instances it creates and hence it is similar to that of FCP. Latency in EXP-e is similar to EXP, since EXP-e only replicates tasks; this improves the throughput but does not alter the latency. Though the performance benefit of our algorithm is limited when CCR is 0.1, as CCR is increased we record greater improvements over existing schemes.

With respect to utilization ratios, we find that MOS uses fewer resources than FCP and generates lower latency schedules. Though resource utilization of EXP is lower, it generates much larger latencies than MOS. When CCR is 0.1, we found that increasing k did not change the performance results. Also, task duplication did not show a significant benefit. This is expected as the communication values are an order of magnitude smaller than computation for CCR=0.1.

Figure 5 shows the relative performance of the schemes on 32 processors with CCR set to 1 and k set to 4 and 8. We find that as CCR increases, there are more instances (T_{max} and 0.75*T_{max}) where FCP, EXP and their modified versions do not meet the throughput constraint, while MOS always does. In addition, the performance benefit of MOS over FCP and EXP is more. At $T = 0.25 * T_{max}$ and $k = 4$, FCP and EXP generate latencies that are 30% and 37% longer than that of MOS respectively, on the average. This is because, MOS intelligently avoids heavy communication costs through techniques like duplication and clustering. Though FCP minimizes communication costs in some capacity by mapping copies of tasks to processors that yield their least completion time, it would still incur the cost when the processor to which the parent task is mapped is heavily loaded (as mapping the task to this processor would cause a larger completion time). EXP does not replicate and cannot cluster heavy tasks that also have a huge communication cost. The modified versions of the schemes also cannot completely avoid the communication overheads as they only replicate tasks. Please note that the average latency ratios of the modified versions is different from that of the original schemes, since there are some cases where the



(a)



(b)

Figure 6. Performance on synthetic graphs on 32 processors for CCR=10 (a)k=4 (b)k=8. (The missing bars indicate that the corresponding algorithm could not meet the throughput requirement.)

original schemes do not meet the throughput constraint while the modified versions do. As k increases, we find a slight increase in the performance benefit of MOS over other schemes. This is probably due to a greater chance for useful task duplication, as the broadcasting of input data to the duplicates is less expensive, since a processor can communicate with more neighbors simultaneously. When CCR is 10, we see similar trends (Figure 6).

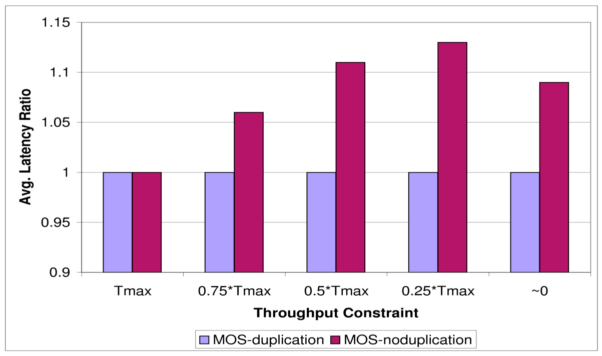
Figure 7 studies the benefit of duplication in yielding lower latency schedules. When CCR=1, duplication is able to achieve up to 13% lower latencies. Since T_{max} is achieved when every every single processor is fully utilized by non-duplicate copies of the tasks, for this value of throughput requirement, the version of the algorithm that allows duplication gives the same result as the one without duplication. With decreasing values of throughput requirement, improvement of duplication increases making a peak around 0.50*T_{max}. When T is further relaxed, not many edges need to be zeroed-in (i.e the incident tasks clustered together) to meet the constraint. Hence clustering does not have as much of an adverse effect in increasing the latency (due to reduced task-parallelism) as for larger throughput requirements. Hence the relative improvement of duplication diminishes. As explained above, as k increases, the possibility of useful duplication increases.

As stated in Section 4, MOS generates a pipelined and a non-pipelined schedule, and chooses the one that yields the lower latency while meeting the throughput requirement. In all our experiments, we found that a non-pipelined schedule proves beneficial in only a few cases when the throughput constraint is negligible.

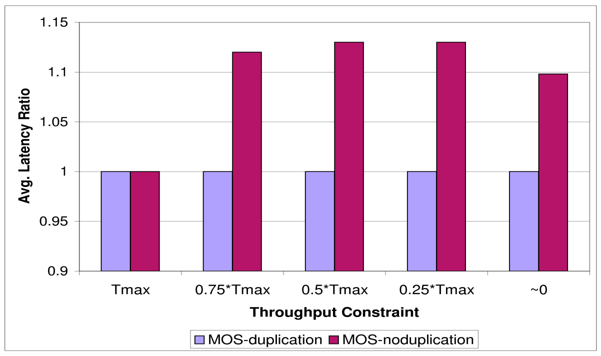
5.2 Application Task Graphs

We evaluated the schemes using application task graphs in the domains of computer vision, multimedia and medical imaging. Due to space constraints, we present results for only two applications. We assumed the target system to be a cluster of 2.4 GHz machines connected by a 10 GigE network.

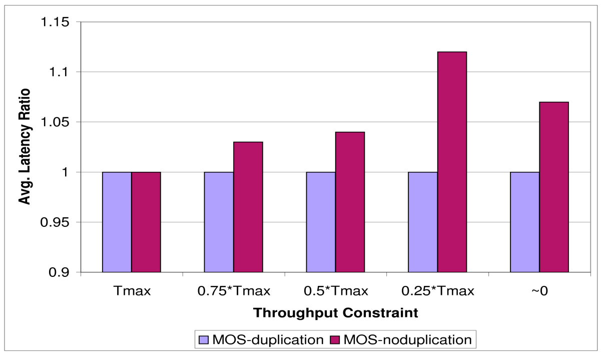
Tables 1 and 2 shows the performance for the Darpa Vision Benchmark (DVB) [27], which contains 20 tasks in 8 levels, with upto 4 tasks per level. The communication costs are about 1 to 3 times the computation costs. For $T < 0.25 * T_{max}$, FCP, FCP-e, EXP and EXP-e do not meet the constraint and hence we do not show the values. Further, when FCP and EXP do not meet the throughput requirement,



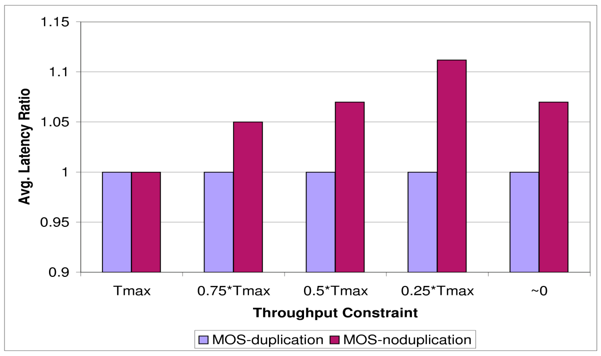
(a)



(b)



(c)



(d)

Figure 7. Benefit of task duplication for (a)CCR=1, k=4 (b) CCR=1, k=8 (c) CCR=10, k=4 (d) CCR=10, k=8.

they generate schedules with throughput atleast 50% less than the constraint. In cases where they satisfy T, MOS produces schedules with shorter latencies and lower resource utilization than FCP. When T is negligible and k is 8, FCP not only uses 25% more processors than MOS but also generates schedule that has 11% longer latency. EXP also produces longer latencies than MOS, up to 22% longer for negligible T and $k = 8$. The increased performance benefit of MOS with larger k is due to greater possibility of duplication. In DVB, as communication costs are significant and the task graph is wide (the source task has multiple children), duplication is effective.

The second application is a video-based surveillance application [2] that analyzes multiple camera feeds from a region to extract information to detect suspicious activity. This task graph contains 16 tasks and is compute intensive with CCR ≈ 0.01. Tables 3 and 4 show the latency ratio and utilization ratio respectively on 32 processors for this application. When the throughput requirement is large, only MOS meets the requirement. In cases where FCP and FCP-e meet the constraint, they generate schedules with similar latency as MOS. As described earlier, when the throughput constraint is negligible, EXP and EXP-e map all tasks to the same task-cluster and hence show a larger latency. With respect to resource utilization, the resource usage of MOS is up to 13% less than the other approaches. We found that varying k did not have an impact on the performance results and task duplication did not help in further latency minimization as this application is compute intensive. Moreover, this application's latency is

k	T	MOS	FCP	FCP-e	EXP	EXP-e
2	0.25*T_{max}	1	-	1.02	-	-
	$\approx$0	1	1.04	1.04	1.14	1.14
4	0.25*T_{max}	1	-	1	-	-
	$\approx$0	1	1.08	1.08	1.20	1.20
8	0.25*T_{max}	1	-	1	-	-
	$\approx$0	1	1.11	1.11	1.22	1.22

Table 1. Performance of Darpa Vision Benchmark on 32 processors for k = 2, 4, 8: Latency Ratio. (The missing values indicate that the corresponding algorithm could not meet the throughput requirement.)

k	T	MOS	FCP	FCP-e	EXP	EXP-e
2	0.25*T_{max}	0.34	-	1	-	-
	$\approx$0	0.09	0.47	0.47	0.03	1
4	0.25*T_{max}	0.5	-	1	-	-
	$\approx$0	0.25	0.47	0.47	0.03	1
8	0.25*T_{max}	0.41	-	1	-	-
	$\approx$0	0.22	0.47	0.47	0.03	1

Table 2. Performance of Darpa Vision Benchmark on 32 processors for k = 2, 4, 8: Utilization Ratio. (The missing values indicate that the corresponding algorithm could not meet the throughput requirement.)

bounded by a heavy task and hence there is not much scope for latency improvement.

The second application is an MPEG video compression application [18]. Due to frame encoding dependences, the MPEG frames have to processed in order of arrival. Hence, replication is not possible. We assumed the throughput constraint to be the reciprocal of the weight of the largest task. Though replication is not possible, the input frames can be divided into N segments, that can be processed in parallel. Figure 8 shows the latency and utilization ratio of the MPEG application on 32 processors, as we vary the number of divisions from 2 to 16. We find that FCP and MOS generate schedules with similar latencies, but MOS has up to 28% lower resource utilization. Though EXP shows lower utilization, it generates schedules with 12%-36% longer latencies than MOS or FCP. The MPEG application is also compute intensive (CCR less than 0.001) and hence did not show variations in performance results with k, or a benefit of task duplication.

T	MOS	FCP	FCP-e	EXP	EXP-e
T_{max}	1	-	-	-	-
0.75*T_{max}	1	-	-	-	1.01
0.5*T_{max}	1	1	1	-	1.01
0.25*T_{max}	1	1	1	-	1.01
$\approx$0	1	1	1	2	2

Table 3. Performance of Video-Based Surveillance application on 32 processors when $k = 4$: Latency Ratio. (The missing values indicate that the algorithm could not meet the throughput requirement).

T	MOS	FCP	FCP-e	EXP	EXP-e
T_{max}	1	-	-	-	-
0.75*T_{max}	0.94	-	-	-	1
0.5*T_{max}	0.78	0.91	0.91	-	1
0.25*T_{max}	0.53	0.66	0.66	-	1
≈0	0.35	0.47	0.47	0.03	1

Table 4. Performance of Video-Based Surveillance application on 32 processors when $k = 4$: Utilization Ratio. (The missing values indicate that the algorithm could not meet the throughput requirement).

Divisions	MOS	FCP	EXP
2	1	1	1.21
4	1	1	1.36
8	1	1	1.35
16	1	1	1.12

(a)

Divisions	MOS	FCP	EXP
2	0.13	0.22	0.09
4	0.25	0.41	0.22
8	0.5	0.78	0.47
16	1	1	1

(b)

Figure 8. Performance of MPEG video compression on 32 processors when $k = 4$ (a) Latency Ratio, (b) Utilization Ratio.

We also evaluated the schemes using a workflow from medical imaging - Placenta Workflow. The stages in this workflow are described in [1]. The execution times of the tasks in this workflow was obtained by profiling them on a dual processor Opteron 250 (single core) with 8GB of RAM and 2x250GB SATA disk. The network bandwidth was assumed to be 10 Gbps Ethernet. Tables 5 and 6 show the performance results. We find similar trends in the performance as for the other applications. FCP and MOS generated similar latencies, while EXP created longer schedules. MOS uses less resources than FCP. Again variation with k or benefit of duplication was not observed as the latency is bounded by heavy stages like NPoint Correlation and Slide Registration that took almost three orders of magnitude longer execution times than other stages.

T	MOS	FCP	FCP-e	EXP	EXP-e
T_{max}	1	-	-	-	-
0.75*T_{max}	1	1	1	-	1
0.5*T_{max}	1	1	1	-	1
0.25*T_{max}	1	1	1	-	1
≈0	1	1	1	1.69	1.69

Table 5. Performance of Placenta workflow on 32 processors: Latency Ratio. (The missing values indicate that the corresponding algorithm could not meet the throughput requirement).

T	MOS	FCP	FCP-e	EXP	EXP-e
T_{max}	1	-	-	-	-
0.75*T_{max}	0.91	1	1	-	1
0.5*T_{max}	0.66	0.75	0.75	-	1
0.25*T_{max}	0.41	0.5	0.5	-	1
≈0	0.13	0.28	0.28	0.03	1

Table 6. Performance of Placenta workflow on 32 processors: Utilization Ratio. (The missing values indicate that the corresponding algorithm could not meet the throughput requirement).

6 Optimizing Throughput under Latency Constraint

The solution described in Section 4 for generating schedules that optimize latency while meeting throughput requirements, can be used to solve the inverse problem of optimizing throughput while meeting latency constraints. In this section, we describe an approach that applies binary search techniques combined with a bounded look-ahead using the algorithm proposed in Section 4 to generate schedules that optimize throughput under latency constraints.

Proposition 2 *For a given workflow and a fixed number of processors, the optimal latency achieved, follows a monotonically increasing relation with the throughput constraint, i.e if L_a is the optimal latency given a throughput constraint T_a, $0 < T_a \leq T_{max}$, and L_b is the optimal latency for a throughput constraint T_b, $0 < T_b \leq T_{max}$, then $T_b \geq T_a \iff L_b \geq L_a$. T_{max}, the theoretical maximum throughput achievable can be computed by applying theorem 1.*

The proof for the above proposition is quite straightforward and the monotonic relationship between latency and throughput justifies the application of binary search algorithm. Hence, starting with the interval $[\approx 0 - T_{max}]$ and the corresponding latencies generated by our heuristics, based on the given latency constraint, we can do a recursive binary search to obtain the maximal throughput schedule that satisfies a given latency constraint. However, as the latency corresponding to a given throughput constraint is obtained based on a heuristic (and may not the optimal latency), the function between the throughput constraint and latency is not guaranteed to be monotonically increasing. Hence, it is possible that the binary search may be trapped in a local optima. To avoid this, we incorporate a bounded look-ahead mechanism into our search technique.

Algorithm 4 describes our approach to optimize throughput under latency constraints. We call this algorithm MOS^{-1} to denote that this focuses on the inverse of the problem addressed by MOS. Given a latency constraint, MOS^{-1} outputs a schedule that maximizes the throughput while meeting the given latency constraint. If the input latency constraint is less than the minimum latency achieved by MOS, the algorithm outputs that it cannot meet the given constraint (Steps 6-8). If the latency constraint is greater that that returned by MOS when given a throughput constraint of T_{max}, MOS^{-1} returns this task mapping (Steps 13-14). Otherwise, MOS^{-1} does a binary search starting with the interval $[\approx 0, T_{max}]$ (Steps 15, 25). Bounded look-aheads are introduced as needed in the search technique to avoid local optima. $BoundedLA_{thr}(T, L)$ is a function that searches upto a limited number of tries, for the maximum throughput in the interval $[T, 2 \times T]$ that satisfies the latency constraint L (by applying MOS) and returns this throughput and the corresponding task mapping. $BoundedLA_{lat}(T, L)$ is a function that searches

Algorithm 4 Algorithm for Throughput Optimization under Latency Constraint

```
1: function $MOS^{-1}(G, L)$                    ▷ $G$: workflow DAG, $L$: latency constraint
2:     $T_a \leftarrow \approx 0$; $< M_a, L_a > \leftarrow MOS(G, T_a)$
3:     $T_b \leftarrow T_{max}$; $< M_b, L_b > \leftarrow MOS(G, T_b)$
4:     if $L \leq L_a$ then
5:         $< M_a, L_a > \leftarrow BoundedLA_{lat}(T_a, L_a)$        ▷ $BoundedLA_{lat}$ searches upto a limited number
               of tries, for the minimum latency returned by $MOS$ given throughput constraints in the interval
               $[T_a, 2 \times T_a]$ and returns this minimum latency and the corresponding task mapping.
6:         if $L < L_a$ then
7:             Cannot meet latency constraint
8:             return
9:         else
10:            $< M, T > \leftarrow BoundedLA_{thr}(T_a, L)$ ▷ $BoundedLA_{thr}$ searches upto a limited number of tries,
                   for the maximum throughput in the interval $[T_a, 2 \times T_a]$ that satisfies the latency constraint $L$
                   (by applying MOS) and returns this throughput and the corresponding task mapping.
11:            return $M, T$
12:    else
13:        if $L \geq L_b$ then
14:            return $M_b, T_b$
15:    while $T_a < T_b$ do
16:        $T' \leftarrow \frac{(T_a+T_b)}{2}$
17:        $< M', L' > \leftarrow MOS(G, T')$
18:        if $L \geq L'$ then
19:            $T_a \leftarrow T'$; $L_a \leftarrow L'$
20:        else
21:            $< M', L' > \leftarrow BoundedLA_{lat}(T', L')$
22:            if $L < L'$ then
23:                $T_b \leftarrow T'$; $L_b \leftarrow L'$
24:            else
25:                $T_a \leftarrow T'$; $L_a \leftarrow L'$
26:    $< M, T > \leftarrow BoundedLA_{thr}(\frac{(T_a+T_b)}{2}, L)$
27:    return $M, T$
```

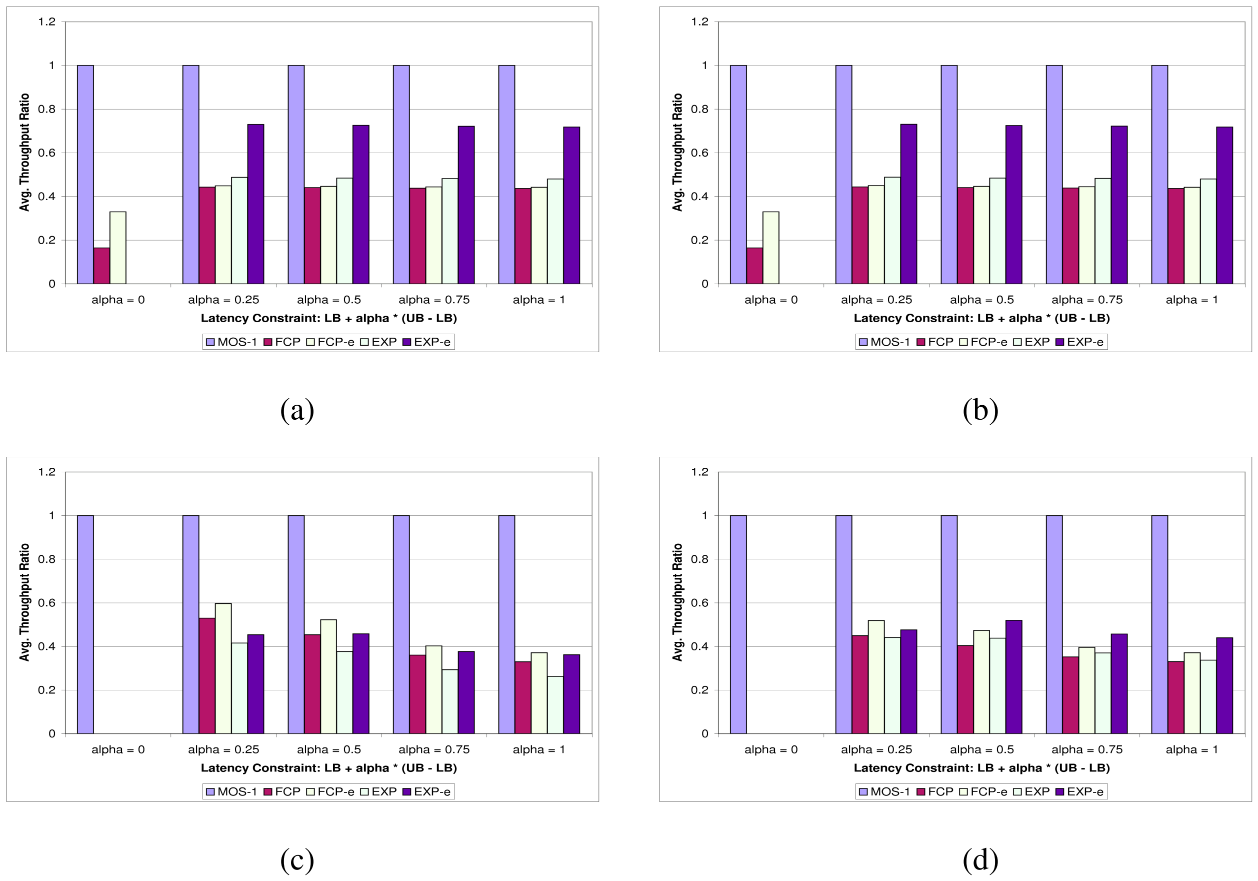


Figure 9. Performance on synthetic graphs for MOS^{-1} on 32 processors (a)CCR=0.1, k=4 (b) CCR=0.1, k=8 (c) CCR=1, k=4 (d) CCR=1, k=8.

upto a limited number of tries, for the minimum latency returned by MOS given throughput constraints in the interval $[T, 2 \times T]$ and returns this minimum latency and the corresponding task mapping.

Figure 9 evaluates our algorithm for optimizing throughput given latency constraints. The lower bound on latency is taken as the critical path length of the given task graph assuming that all edges have zero weight 4. We assume the upper bound on latency to be the sum of all task weights. This would be the latency achieved by mapping all tasks in G to a processor and replicating as many times as number of processors available. This task mapping achieves the maximum possible throughput T_{max} 1. We vary the latency constraint in steps of 0.25 between the lower and upper bounds. To obtain solutions using FCP and EXP and their modified versions, we applied the same binary search algorithm as described in MOS^{-1}, but used FCP and EXP respectively to obtain the schedules with minimal latency given a throughput constraint, instead of MOS. However as EXP has the tendency to map all tasks to the same task-cluster when given negligible throughput constraints (as described in Section 5), we had to have larger look-aheads to avoid local optima.

As seen in Figure 9, our algorithm, MOS^{-1} generates schedules with larger throughputs (upto 60% larger) than other schemes while meeting the latency constraints. In addition, there are cases where MOS^{-1} meets the latency constraints, while the other schemes fail. Please note that when the latency constraint is equal to the lower bound on the latency, it may not always be feasible to obtain schedules that meet this constraint. Hence, in such cases, MOS^{-1} will fail to meet the latency constraint.

For CCR=0.1, we find that varying k does not alter the trends. This is similar to what is observed in Section 5. For larger communication costs, i.e CCR=1, we find that increasing k improves the relative performance of MOS^{-1} with respect to FCP and FCP-e. This is because, while MOS^{-1} is able to generate schedules with greater throughput as a larger k implies more parallel communications, the performance of FCP and FCP-e remains the same, as we compute these, always assuming a fully multi-port model (explained in detail in Section 5). On the other hand, EXP and EXP-e is able to take advantage of the larger k value and hence their relative performance improves.

7 Conclusions

This work presents heuristics for scheduling application workflows with stringent performance requirements. Through co-ordinated leveraging of pipelined, task and data parallelism and use of techniques like task duplication, the proposed algorithm, MOS, minimizes the latency of streaming workflows, while satisfying strict throughput requirements. We also describe a binary search based algorithm using MOS, that optimizes throughput of streaming workflows while meeting latency constraints. Evaluation using synthetic and application task graphs indicate that our heuristic is always guaranteed to meet the throughput requirement and generates schedules with lower latencies than existing approaches, while using lesser resources. When applied to optimize throughput given latency constraints, our algorithm generates schedules with larger throughput than existing approaches. Our experimental results also provide the following insights regarding the benefit of task duplication: a) Duplication is beneficial (generates up to 18% lower latency schedules) for workflows that are wide (large number of concurrent tasks and high average out-degree) and have comparable communication and computation costs, b) Communication system with larger value of k can see greater benefits of duplication, and c) Workflows that are compute intensive and those that have communication costs at least one order of magnitude greater than computation benefit less from task duplication.

References

[1] The placenta image analysis pipeline. http://bmi.osu.edu/∼vijayskumar/placenta1.htm.

[2] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: A scheduling heuristic for streaming application on the grid. In *Proceedings of the 13th annual Multimedia Computing and Networking Conference*, San Jose, CA, Jan 2006.

[3] B. Agarwalla, N. Ahmed, D. Hilley, and U. Ramachandran. Streamline: scheduling streaming applications in a wide area environment. *Multimedia Syst.*, 13(1):69–85, 2007.

[4] I. Ahmad and Y.-K. Kwok. On exploiting task duplication in parallel program scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 9(9):872–892, September 1998.

[5] A. Benoit, H. Kosch, V. Rehn-Sonigo, and Y. Robert. Bi-criteria pipeline mappings for parallel image processing. Technical Report LIP RR-2008-02, 2008.

[6] A. Benoit, H. Kosch, V. Rehn-Sonigo, and Y. Robert. Optimizing latency and reliability of pipeline workflow applications. Technical Report LIP RR-2008-12, 2008.

[7] A. Benoit, V. Rehn-Sonigo, and Y. Robert. Multi-criteria scheduling of pipeline workflows. Technical Report LIP RR-2007-32, 2007.

[8] A. Benoit and Y. Robert. Mapping pipeline skeletons onto heterogeneous platforms. Technical Report LIP RR-2006-40, 2006.

[9] A. Benoit and Y. Robert. Complexity results for throughput and latency optimization of replicated and data-parallel workflows. Technical Report LIP RR-2007-12, 2007.

[10] D. Bozdag, U. Catalyurek, and F. Ozguner. A task duplication based bottom-up scheduling algorithm for heterogeneous environments. In *Proceedings of the 15th Heterogeneous Computing Workshop*. IEEE Computer Society, 2006.

[11] D. Bozdag, F. Ozguner, E. Ekici, and U. V. Catalyurek. A task duplication based scheduling algorithm using partial schedules. In *Proceedings of the 2005 International Conference on Parallel Processing*, pages 630–637, 2005.

[12] L. Chen and G. Agrawal. Resource allocation in a middleware for streaming data. In *MGC '04: Proceedings of the 2nd workshop on Middleware for grid computing*, pages 5–10, New York, NY, USA, 2004. ACM.

[13] L. Chen and G. Agrawal. A static resource allocation framework for grid-based streaming applications: Research articles. *Concurr. Comput. : Pract. Exper.*, 18(6):653–666, 2006.

[14] A. Choudhary, W. Lio, D. Weiner, P. Varshney, R. Linderman, and M. Linderman. Design, implementation and evaluation of parallel pipelined stap on parallel computers. In *Proceedings of the 12th. International Parallel Processing Symposium*, page 220, Washington, DC, USA, 1998. IEEE Computer Society.

[15] A. N. Choudhary, B. Narahari, D. M. Nicol, and R. Simha. Optimal processor assignment for a class of pipelined computations. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):439–445, 1994.

[16] Y. Chung and S. Ranka. Application and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed-memory multiprocessors. *Proceedings of Supercomputing*, pages 512–521, November 1992.

[17] J. Colin and P. Chretienne. C.p.m. scheduling with small computation delays and task duplication. *Operations Research*, pages 680–684, 1991.

[18] F. Guirado, A.Ripoll, C. Roig, and E. Luque. Optimizing latency under throughput requirements for streaming applications on cluster execution. In *Proceedings of the IEEE Intl. Conf. on Cluster Computing*, 2005.

[19] S. L. Hary and F. Ozguner. Precedence-constrained task allocation onto point-to-point networks for pipelined execution. *IEEE Transactions on Parallel and Distributed Systems*, 10(8):838–851, 1999.

[20] J. Jonsson and J. Vasell. Real-time scheduling for pipelined execution of data flow graphs on a realistic multiprocessor architecture. In *Proceedings of the 1996 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 6, pages 3314–3317, 1996.

[21] B. Kruatrachue and T. Lewis. Grain size determination for parallel processing. *IEEE Software*, 5(1):23–32, January 1988.

[22] V. S. Kumar, B. Rutt, T. Kurc, U. Catalyurek, J. Saltz, S. Chow, S. Lamont, and M. Martone. Imaging and visual analysis—large image correction and warping in a cluster environment. In *Proceedings of the 2006 ACM/IEEE conf. on Supercomputing*, page 79, 2006.

[23] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[24] M. Lee, W. Liu, and V. K. Prasanna. A mapping methodology for designing software task pipelines for embedded signal processing. In *Proceedings of the Workshop on Embedded HPC Systems and Applications of IPPS/SPDP*, pages 937–944, 1998.

[25] C. Papadimitriou and M. Yannakakis. Towards an architecture independent analysis of parallel algorithms. *SIAM Journal of Computing*, 19:322–328, April 1990.

[26] B. Shirazi, H. Chen, and J. Marquis. Comparative study of task duplication static scheduling versus clustering and non-clustering techniques. *Concurrency: Practice and Experience*, 7(5):371–390, August 1995.

[27] S. B. Shukla and D. P. Agrawal. Scheduling pipelined communication in distributed memory multiprocessors for real-time applications. *ACM SIGARCH Computer Architecture News*, 19(3), 1991.

[28] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the grid. In *Proceedings of the 2002 ACM/IEEE conf. on Supercomputing*, pages 1–18, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[29] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *Proceedings of the 8th ACM Symposium on Parallel Algorithms and Architecture*, pages 62–71, New York, NY, USA, 1996. ACM Press.

[30] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *ACM/IEEE International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Oct 2005.

[31] K. Vallerio. Task graphs for free. http://ziyang.ece.northwestern.edu/tgff/maindoc.pdf (2003).

[32] N. Vydyanathan, U. Catalyurek, T. Kurc, P. Sadayappan, and J. Saltz. An approach for optimizing latency under throughput constraints for application workflows on clusters. Technical Report OSU-CISRC-1/07-TR03, The Ohio State University, 2007.

[33] N. Vydyanathan, Ü. V. Çatalyürek, T. M. Kurç, P. Sadayappan, and J. H. Saltz. Toward optimizing latency under throughput constraints for application workflows on clusters. In *13th Intl. Euro. Conf. on Par. and Dist. Computing*, pages 173–183, 2007.

[34] M. Yang, T. Gandhi, R. Kasturi, L. Coraror, O. Camps, and J. McCandless. Real-time obstacle detection system for high speed civil transport supersonic aircraft. In *Proceedings of the IEEE National Aerospace and Electronics Conference*, pages 595–601, 2000.

[35] M.-T. Yang, R. Kasturi, and A. Sivasubramaniam. A pipeline-based approach for scheduling video processing algorithms on now. *IEEE Transactions on Parallel and Distributed Systems*, 14(2):119–130, 2003.