Adding Data Parallelism to Streaming Pipelines for Throughput Optimization

Peng Li, Kunal Agrawal, Jeremy Buhler, Roger D. Chamberlain
Department of Computer Science and Engineering
Washington University in St. Louis
St. Louis, MO 63130
{*pengli, kunal, jbuhler, roger*}*@wustl.edu*

***Abstract*—The streaming model is a popular model for writing high-throughput parallel applications. A streaming application is represented by a graph of *computation stages* that communicate with each other via FIFO *channels*. In this paper, we consider the problem of mapping *streaming pipelines* — streaming applications where the graph is a linear chain — onto a set of computing resources in order to maximize its throughput. In a parallel setting, subsets of stages, called *components*, can be mapped onto different computing resources. The throughput of an application is determined by the throughput of the slowest component. Therefore, if some stage is much slower than others, then it may be useful to *replicate* the stage's code and divide its workload among two or more replicas in order to increase throughput. However, pipelines may consist of some replicable and some non-replicable stages. In this paper, we address the problem of mapping these *partially replicable streaming pipelines* onto both homogeneous and heterogeneous platforms so as to maximize throughput.**

We consider two types of platforms, *homogeneous platforms* — where all resources are identical, and *heterogeneous platforms* — where resources may have different speeds. In both cases, we consider two network topologies — unidirectional chain and clique. We provide polynomial-time algorithms for mapping partially replicable pipelines onto unidirectional chains for both homogeneous and heterogeneous platforms. For homogeneous platforms, the algorithm for unidirectional chains generalizes to clique topologies. However, for heterogeneous platforms, mapping these pipelines onto clique topologies is NP-complete. We provide heuristics to generate solutions for cliques by applying our chain algorithms to a series of chains sampled from the clique. Our empirical results show that these heuristics rapidly converge to near-optimal solutions.

I. Introduction

The ***streaming computation model*** has received considerable attention in recent years, as it can exploit task parallelism, data parallelism, and especially pipelined parallelism to speed up computations. Streaming is used to express high-throughput applications such as audio and video processing, biological sequence or astrophysics data analysis, and financial modeling. A streaming computation is a directed graph with ***computational stages*** (vertices) connected by FIFO ***channels*** (edges). Each stage runs a specified computation, which repeatedly receives data on its incoming channels (from its predecessors), computes on the data, and sends output data on its output channels. In this work, we will focus on mapping and scheduling ***streaming pipelines*** — computations with linear chain topologies. Pipeline topologies are common for streaming applications.

Given a streaming pipeline and a platform consisting of a set of computing resources (i.e., processors) connected via a network, a ***mapping*** algorithm is responsible for deciding which stage runs on which resource. A platform is ***homogeneous*** if all its resources are identical or ***heterogeneous*** if different resources have different computational capacities. Because the various stages of the pipeline must communicate with each other, the topology of the platform plays a crucial role in determining feasible mappings. We consider two types of topologies: ***unidirectional chain***, where resources are connected in a linear fashion through one-way channels; and ***clique***, where all resources are connected to all others with bi-directional channels. In this paper, we consider the problem of mapping streaming applications onto both homogeneous and heterogeneous platforms connected in both chain and clique fashion.

A common goal of streaming computation mapping algorithms is to maximize ***throughput***, which is defined as the number of incoming data items processed per unit time in the steady state of the computation. Even if we map each stage of a streaming pipeline to a different resource, throughput is limited by the slowest stage.

In this paper, we focus on ***stage replication*** in order to overcome barriers to higher throughput. Replicating a stage means making more than one copy of the stage, then running these copies on different resources, thereby dividing the workload of this stage. Replication introduces data parallelism into a streaming application. We consider only replication strategies that are entirely safe; that is, the streaming application after replication should give exactly the same outputs in exactly the same order as the original application.

We consider the general case of a pipeline where some stages can safely be replicated, while others cannot. If a stage keeps internal state and updates that state during its computation, it is a ***stateful*** stage; otherwise,

it is ***stateless***. (Note that a "stateless" stage might still keep static state, which does not change during computation.) If a stateful stage were replicated, different copies of the stage would need to coordinate with each other in order to maintain their states and so compute correctly, which could be expensive. In addition, it could be impossible to correctly maintain state with replication for some types of stage. In contrast, stateless states can be replicated with almost no overhead, since the computation for a given input does not depend on any previous computation; hence, all data items can be processed independently in parallel. In this paper, we assume all stateful stages to be non-replicable and all stateless stages to be replicable.

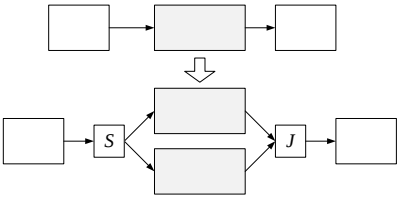

Figure 1. Replicating a stage using split and join nodes S and J.

In practice, after a stage is replicated, a split node and a join node are inserted into the streaming pipeline, as Figure 1 shows. The split node collects data from upstream and distributes them to the stage's replicas, while the join node collects data from the replicas and sends them downstream. A split node might not distribute data evenly to all replicas; instead, some replicas might receive more data and thus do more work than others to achieve load balancing. In this paper, to simplify the computation model, we assume that the computational costs of split and join nodes are negligible and so ignore their overhead when choosing a mapping. We also assume that communication can be fully overlapped with computation and so do not consider communication overhead.

In this paper, we devise mapping strategies for ***partially replicable pipelines***, where certain stages are replicable while others are not. Our contributions are as follows:

- For homogeneous platforms, we provide a polynomial-time algorithm for mapping partially replicable pipelines onto unidirectional chain topologies. It turns out that cliques are not more powerful, and the same algorithm works for them.
- For heterogeneous platforms, we provide a polynomial-time algorithm for mapping onto chains. Mapping onto cliques is NP-complete even for non-replicable pipelines [1], [2], [3].
- We provide heuristics for mapping onto heterogeneous cliques, using our algorithm for chains as a subroutine. Our empirical results indicate that these heuristics perform reasonably well in practice.
- While our algorithm for mapping onto heterogeneous chains is polynomial, it has a high complexity. We therefore provide an approximation algorithm that is near-linear in the sizes of both the pipeline and the platform.

II. Related Work

Most prior works on maximizing throughput of pipelines do not consider replication. Bokhari solved the throughput optimization problem for pipeline mapping by finding a minimum bottleneck path in a layered graph that contains all information about application modules [4]. Hansen et al. later improved Bokhari's solution using dynamic programming [5]. For the case when processors are homogeneous, many efficient algorithms have been proposed, such as [6], [7], and [3]. However, when some stages are replicable, these algorithms no longer work as they rely on finite mapping boundaries. Some recent works provide complexity results for mapping pipelines onto homogeneous and heterogeneous platforms, both with and without communication costs [1], [2]. Agrawal et al. provided mapping solutions for the problem when stages can filter data [8].

Replication has been considered in some limited cases. Subhlok et al. considered a model where every task can be perfectly parallelized on homogeneous processors and provided mapping solutions for optimizing throughput [9] as well as solutions for latency-throughput trade-offs [10]. Recently, Kudlur et al. and Cordes et al. used integer linear programming to extract data parallelism from streaming pipelines [11], [12]. While they considered replication, they all assumed that the work of replicated stages should be *evenly* divided and assigned to replicas. In contrast, not all replicas necessarily have the same amount of work in our model.

Several existing works have tried to maximize pipeline throughput with empirical approaches. For example, Gordon et al. developed heuristics to exploit data parallelism in streaming programs by fusing stateless filters [13]. Wang et al. proposed a machine learning-based approach to mapping streaming applications on multi-cores [14]. These approaches did not provide optimality guarantees. Our method differs from them in that we try to guarantee provable optimality.

III. Problem Formulation

In this section, we precisely formulate the problem of throughput-optimal mapping for partially replicable pipelines. We also describe the target platform to which a pipeline may be mapped, whose characteristics determine the complexity of the mapping problem.

Streaming Pipelines

A linear pipeline is a sequence of m stages $S_1 \ldots S_m$, where each stage S_i is connected via a communication channel to the next stage S_{i+1}. Each stage has a characteristic ***work*** $W(S_i)$, which is the time taken to execute the stage (each time it fires) on some fixed benchmark processor. In addition, each stage S_i has (integral) ***input*** and ***output rates*** $in(S_i)$ and $out(S_i)$ specifying, respectively, the number of data items consumed from its incoming edge and the number of items emitted onto its outgoing edge each time it fires. We assume, without loss of generality, that the input rate $in(S_1)$ of the first stage (the ***source***) is always 1. We assume that the pipeline follows the ***synchronous dataflow*** model [15], where work and the input and output rates remain fixed and are known in advance.

Definition 1. The ***gain*** $g(S_i)$ of stage S_i is the number of times S_i fires every time the source stage S_1 fires. For a linear pipeline,

$$g(S_i) = \prod_{j=2}^{i} (in(S_j)/\ out(S_{j-1})) \ .$$

Definition 2. The ***normalized work*** $w(S_i)$ of stage S_i is the amount of work the stage does, on average, for each input consumed by the source node of the pipeline. From the preceding definitions, we have that

$$w(S_i) = g(S_i) \cdot W(S_i) \ .$$

Note that a stage's normalized work may be greater or less than its work, depending on the gain of the stage.

Replication of Stages

If a stage is stateless, then it can be replicated by adding a split node before the stage and a join node after it. Without loss of generality, we assume that a replicable stage is either at the beginning or the end of the pipeline or has non-replicable stages on both sides. This is due to the fact that consecutive replicable stages can be merged into a single replicable stage [13].

When a stage is replicated, each replica is called a ***node***. The terms "stage" and "node" are interchangeable for non-replicable stages. A node created as a replica of stage S_i has the same input and output rates and the same work as S_i. As noted previously, replication involves the insertion of split and join nodes before and after S_i, which are assumed to do no work and to have input and output rates of 1. A split node distributes its inputs among replicas $c_1, c_2, \ldots$ of S_i, with each replica c_k receiving a fraction f_k of inputs such that $\sum_k f_k = 1$. The gain of a split node is the gain of its predecessor; the gain of replica c_k is $g(S_i) \cdot f_k$; and the gain of the join node is $g(S_i)$. The normalized work for any node is still defined as its gain multiplied by its work.

In formulating our mapping algorithms, we will not explicitly refer to replication but instead will use the concept of ***dividing*** replicable pipeline stages. The work of a replicable stage can be divided among multiple replicas, each of which can be mapped to a different resource. The division need not be even; for example, one replica may receive 80% of inputs to the stage, and so do 80% of its work, while another does only 20%. In principle, a replica may receive any real-valued fraction of the stage's work. In practice, there may be limits on the granularity of work division; however, for long input streams, there may still be hundreds or thousands of distinct ways to divide a stage's work, so the continuous approximation remains useful.

Because we assume that the split and join nodes needed to realize a replicated topology do no work, and we ignore communication costs, we will treat a replicated stage as simply being divided into pieces, each of which does some fraction of the stage's work. We will refer to stages as being ***divisible*** or ***indivisible***, interchangeably with replicable or non-replicable.

Once a pipeline has undergone partial replication, the resulting network of nodes is mapped onto resources. Multiple nodes may be mapped to a single resource, in which case they form a ***component***. Nodes mapped to a single resource are assumed to execute sequentially, so the normalized work of a component is the sum of the normalized works of its constituent nodes.

Optimization Problem

We seek to maximize throughput, which is defined as the average number of inputs consumed by a pipeline per unit time during steady-state computation. The throughput of a pipeline is the inverse of its ***period*** τ, the minimum time the source must wait between consuming one input and the next to ensure that no stage receives data items faster than it can process them. In this paper, we describe algorithms in terms of period minimization, which is equivalent to throughput maximization.

Suppose we have a set of homogeneous resources $P_1 \ldots P_n$, such that the execution time of a stage on one such resource determines its work. If a pipeline is mapped onto these resources (perhaps with partial replication), let $w(P_j)$ be the normalized work of the component executing on P_j. Then we have $\tau = \max_j w(P_j)$. If instead the resources are heterogeneous, each resource P_j has some ***speed*** sp_j, which is defined as a scaling factor relative to the benchmark processor used to quantify work. In this case, we define the ***scaled work*** of resource j to be $w(P_j)/\ sp_j$, and we have $\tau = \max_j w(P_j)/\ sp_j$.

We now formally define the optimization problem addressed in this paper. We are given a linear pipeline with m stages $S_1 \ldots S_m$, each with defined input rate, output

rate, and work. Each stage is labeled as either divisible or indivisible. Our goal is to map this pipeline, possibly with partial replication, onto n resources $P_1 \dots P_n$ with speeds $sp_1 \dots sp_n$, so as to minimize the period τ of the resulting physical realization.

The minimum realizable period for any mapping of a pipeline onto resources depends on the set of feasible mappings, which depends on how resources on the target platform are interconnected. We consider constraints on feasible mappings in the next section.

Feasible Mappings

Two common types of constraint on feasible mappings for streaming pipelines are ***unconstrained mapping*** and ***contiguous*** or ***convex mapping***. In unconstrained mapping, any arbitrary combination of stages may be mapped onto a single resource, while in contiguous mapping, each resource receives a contiguous interval of stages from the pipeline.

Unconstrained mapping permits more choices among mappings, but it has two drawbacks. First, the physical dataflow graph is not acyclic, potentially leading to pitfalls such as deadlocks. Second, communication between resources increases; in contiguous mappings, only the links that cross components lead to physical communication, while for non-contiguous mappings, all links potentially cause communication. Even for multicore machines with no physical links, one can show that contiguous mappings minimize the number of cache misses [16]. Moreover, contiguous mappings are a 2-approximation of unconstrained mappings on homogeneous platforms [2]. Though we do not consider communication overhead in this paper, we will nonetheless focus on contiguous mappings as a practically useful constraint on our solution space.

In the presence of replication, we extend the definition of contiguous mapping as follows. A mapping of nodes in a replicated pipeline to resources is contiguous if for *some* topological ordering of the pipeline's nodes, the mapping is contiguous in the sense described above, as Figure 2a shows. This constraint preserves the relative order of a stage's replicas with the stages before and after them and so ensures that, as for a simple pipeline, the physical dataflow graph resulting from the mapping is acyclic. On the other hand, a mapping that maps stages S_1, S_{21} and S_3 onto the same resource and S_{22} onto another resource would not be contiguous.

When we think about mappings in term of division rather than replication, conceptually, a contiguous mapping places *boundaries* between components at various points in the pipeline, either at one end of a stage or inside a divisible stage. When a boundary is within a divisible stage such that f_1 fraction of the divisible stage is on the left and f_2 is on the right, we create two replicas of the stage, where the first replica is in the left component and receives f_1 fraction of the stage's work, and the other replica is in the right component and receives f_2 fraction of the work.

Target Platform Topology

A pipeline's mapping must be feasible given the physical interconnections among resources on the target platform. In particular, the physical dataflow graph cannot include an edge from P_i to P_j if there is no channel to carry the data on this edge. Moreover, if some channels are unidirectional, the mapping must respect this fact; the graph cannot include an edge from P_i to P_j if the only available channel goes from P_j to P_i.

In this paper, we focus on mappings to target platforms whose resources are interconnected in one of two ways: a ***unidirectional chain***, or a ***clique*** (i.e. all possible bidirectional connections among resources). The unidirectional chain is a simple topology that permits tractable optimization; moreover, it is a natural platform on which to realize pipelines of abstract stages. The clique interconnect is typical of distributed systems today, in which all processing elements are logically fully connected, no matter the actual physical interconnect.

The reader may ask how a pipeline with replicated stages can be mapped onto a unidirectional chain of resources. We assume that, when necessary, data items can be forwarded from earlier to later resources in the chain without communication overhead. Alternatively, if the chain was extracted from a clique topology, there are already forwarding channels that can be used to bypass intermediate resources where necessary. Figure 2b visualizes these two cases.

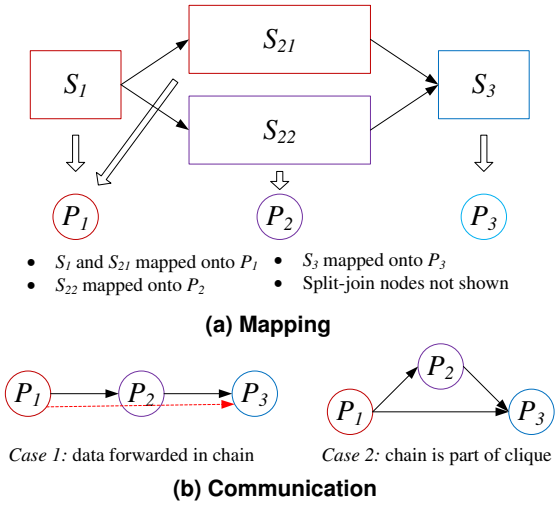

Figure 2. Replicated stages mapped onto unidirectional chains.

For a target with homogeneous resources, there is effectively no difference between the chain and clique

interconnects because every unidirectional chain of a given length embedded within the clique is identical to every other. Hence, we may choose one such chain from the clique arbitrarily. This equivalence among chains does *not* hold for heterogeneous resources, since each chain may consist of resources with distinct speeds arranged in a distinct order. Hence, there are really three versions of the problem for contiguous mappings:

- mapping a pipeline onto (WLOG) a unidirectional chain of homogeneous resources;
- mapping a pipeline onto a unidirectional chain of heterogeneous resources;
- mapping a pipeline onto a clique of heterogeneous resources.

Since we assume that the pipeline follows the synchronous dataflow model, a bounded-memory schedule can be found efficiently based on the mapping. We omit the details in this paper due to space considerations.

IV. Throughput Optimization on Homogeneous Platforms

In this section, we discuss throughput optimization on homogeneous platforms, in which all resources have the same speed. We address mapping onto unidirectional chains, which, as discussed in Section III, extends WLOG to homogeneous cliques.

If all pipeline stages are indivisible, the mapping problem is easily solved in polynomial time via dynamic programming [5]. The general subproblem considers a triple (i, j, k), where pipeline stages i to j inclusive must be mapped onto k contiguous resources in the chain. The number of such subproblems is $O(m^2 n)$.

If, however, some pipeline stages are divisible, we now have an unbounded number of choices for how to divide stages among resources. If we are to transfer the dynamic programming approach to the new problem, we need a basis for limiting the number of choices. To do so, we begin with the following definition.

Definition 3. A mapping of a partially replicable pipeline to resources is a said to be a ***perfectly divided mapping (PDM)*** if each resource's component has the same scaled work. That is, for each resource P_i, $w(P_i)/\,sp_i$ must be the same.

For homogeneous platforms, a PDM implies that each resource's component has the same normalized work. Any given pipeline may or may not have a PDM. For example, Figure 3a shows a pipeline with five stages with normalized works 1, 4, 2, 6, and 1, such that only the second and fourth stages are divisible. This pipeline has a contiguous PDM onto four identical resources, each of which receives normalized work 3.5. The second stage is divided into two pieces of sizes 0.625 and 0.375, while the fourth is divided into pieces of sizes roughly 0.583 and 0.417. By contrast, Figure 3b shows a slightly modified pipeline with normalized works 1, 2, 4, 6, and 1, respectively. This pipeline has *no* contiguous PDM onto four identical resources. In general, it is straightforward to check in time $O(m + n)$ whether a given partially replicable pipeline of m stages has a contiguous PDM onto n resources.

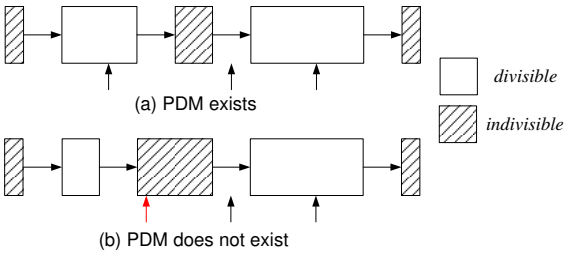

Figure 3. Pipelines with and without perfectly divided mappings onto 4 resources. Length of each stage's box is proportional to its normalized work. Vertical arrows indicate component boundaries.

If a given pipeline has a PDM onto a set of resources, that PDM achieves the minimal period among all mappings and hence is throughput-optimal, since no resource is a bottleneck relative to any other. If such a PDM does *not* exist, we now show that we can effectively subdivide the mapping problem.

Theorem IV.1. (Fixed-boundary Theorem) *If a partially replicable pipeline lacks a PDM for a given set of resources, then some throughput-optimal mapping of the pipeline has an internal component boundary at one end of an indivisible stage.*

Proof: We proceed by contradiction. Suppose the pipeline has no PDM, and let M be a throughput-optimal mapping of it with period τ. If M has an internal boundary at one end of an indivisible stage, we are done. Otherwise, every internal boundary in M lies inside a divisible stage. Since M is not a PDM, there must be some bottleneck stage P_k, i.e. a stage with normalized work τ, that is adjacent to a non-bottleneck stage that does less work. WLOG, assume that P_{k+1} does less work than P_k. Then we can move back the boundary between the components on P_k and P_{k+1}, thereby transferring a nonzero amount of work from the former to the latter.

If P_k held the only component with normalized work τ, we have improved the mapping M, which contradicts its optimality. Otherwise, we repeat the boundary moving operation on the remaining components with normalized work τ until all have been improved, and we again achieve contradiction. ■

Theorem IV.1 shows that if there is no PDM for a given pipeline, then we can subdivide the problem for dynamic programming purposes by finding the

best mapping with a component boundary at each end of each indivisible stage, then keeping the best of these mappings. In more detail, let OPT_DM$[i, j, k]$ denote the period of a throughput-optimal mapping of pipeline stages $i..j$ onto k resources. To compute OPT_DM$[i, j, k]$, we first check if there is a PDM for $i..j$ onto k resources. If so, we are done; otherwise, we enumerate the boundaries of indivisible stages between the start of stage i and the end of stage j. For each such boundary b, we subdivide the stages $i..j$ around b and consider all ways of allocating the k resources to the two resulting subproblems, keeping the best result found. Pseudocode for this method is provided in Algorithm 1.

Algorithm 1: Throughput-Optimal Mapping on Homogeneous Platforms

Input: Pipeline, # resources
Output: Period of throughput-optimal mapping for the pipeline

let m be the number of pipeline stages
let n be the number of resources
return DP_HELPER$(1, m, n)$

Function: DP_HELPER(i, j, k)
Input: Pipeline segment, # resources
Output: Period of throughput-optimal mapping for the segment

if $i = j$ **then**
  if S_i *is indivisible* **then**
    OPT_DM$[i, j, k] \leftarrow w(S_i)$
  else
    OPT_DM$[i, j, k] \leftarrow w(S_i)/k$
else if OPT_DM*$[i, j, k]$ not yet computed* **then**
  if *PDM exists with period* τ **then**
    OPT_DM$[i, j, k] \leftarrow \tau$
  else
    index stage x's boundaries as x and $x + 1$
    let B be boundaries of indivisible stages
    $B_{ij} \leftarrow \{x \mid i < x \leq j \text{ and } x \in B\}$
    OPT_DM$[i, j, k] \leftarrow \infty$
    foreach $b \in B_{ij}$ **do**
      for $r \leftarrow 1$ **to** $k - 1$ **do**
        $\tau_1 \leftarrow$ DP_HELPER$(i, b - 1, r)$
        $\tau_2 \leftarrow$ DP_HELPER$(b, j, k - r)$
        $\tau \leftarrow \max(\tau_1, \tau_2)$
        if $\tau <$ OPT_DM*$[i, j, k]$* **then**
          OPT_DM$[i, j, k] \leftarrow \tau$
return OPT_DM$[i, j, k]$

Given m stages and n resources, there are $O(m^2 n)$ calls to DP_HELPER(). Each call takes time at most $O(mn)$ to compute in addition to its recursive calls, so the total time complexity is $O(m^3 n^2)$.

V. Throughput Optimization on Chain-connected Heterogeneous Platforms

In this section, we extend the results of the previous section to unidirectional chains of heterogeneous resources. We first show that Algorithm 1 can be modified to address this case, then give a practically much faster approximation algorithm to find a solution that is provably near-optimal.

Optimal Mapping Algorithm

For heterogeneous resources, a perfectly divided mapping (PDM) onto resources must take the speed of each resource into account. Nevertheless, it is still feasible to check in time $O(m + n)$ whether a partially replicable pipeline of m stages has a PDM onto a given chain of n resources, since the amount of normalized work to be allocated to each resource is easily computable, and the order of resources is fixed. Moreover, the proof of Theorem IV.1 goes through unaltered, because it merely requires that one be able to move a component boundary so as to remove (scaled) work from one resource and add it to the adjacent resource; the work added and removed need not be equal. Hence, we again conclude that, if no PDM exists, there must be a throughput-optimal mapping with an internal component boundary at one end of an indivisible stage.

The recurrence of Algorithm 1 needs to be modified to account for heterogeneous resources. In the recurrence for the homogeneous case, each subproblem OPT_DM$[i, j, k]$ was parametrized by a range of pipeline stages and a number of resources. Any contiguous interval of k resources yielded an equivalent solution. In the heterogeneous case, mapping to different contiguous intervals of k resources, each with its own speed, results in realizations with different periods.

To account for this extra complexity, we modify the recurrence to compute subproblems OPT_DM$[i, j, p, q]$, where $p..q$, $1 \leq p \leq q \leq n$, is a contiguous interval of resources onto which we map stages $i..j$. The full problem is now to compute OPT_DM$[1, m, 1, n]$, and the subproblems are of the form OPT_DM$[i, j, p, q]$, where the optimal period is potentially different for each $p..q$. Each call to DP_HELPER() still considers $O(mn)$ cases, but the total number of subproblems is now $O(m^2 n^2)$, for a total running time of $O(m^3 n^3)$.

Fast $(1 + \epsilon)$-Approximation Algorithm

The time complexity of the dynamic programming algorithm for throughput-optimal mappings, particularly in the heterogeneous case, is high, making it impractical for large m and/or n. In this section, we describe an asymptotically faster approximation algorithm that obtains a mapping whose period is within a factor $1 + \epsilon$

of the optimum, for any desired ϵ. The algorithm is based on the dual approximation scheme [17].

Consider a pipeline of m stages $S_1 \dots S_m$ mapped to a set of (in general heterogeneous) resources $P_1 \dots P_n$. We will show how to quickly answer the question "does the pipeline have a mapping to these resources with period at most τ?" Given this test as a subroutine, we can approximate the actual optimal period τ^* to within any multiplicative factor $1+\epsilon$ as follows.

1) First, observe that $\tau^* \geq \hat{\tau}$, where
$$\hat{\tau} = \frac{\sum_i w(S_i)}{\sum_j sp_j}.$$
2) Next, use exponential search, starting with $\tau = \hat{\tau}$ and doubling τ each time, to find the first period $\bar{\tau}$ for which the test succeeds. We know that $\bar{\tau} \leq 2\tau^*$.
3) Finally, use binary search on the interval $(\bar{\tau}/2, \bar{\tau}]$ to reduce the difference between the greatest period for which the test is known to fail and the least period τ_u for which it is known to succeed to at most $\epsilon \cdot \hat{\tau}$. Return τ_u as the estimate of τ^*.

Observe that the final upper bound τ_u is at most
$$\tau^* + \epsilon \cdot \hat{\tau} \leq (1+\epsilon)\tau^*.$$

We now develop the test for whether a pipeline can be mapped to a given set of resources with period at most τ. We use the following greedy algorithm. Starting from the beginning of the pipeline, move the first component's right boundary to the right (i.e. downstream) until the normalized work allocated to the first resource is $\tau \cdot sp_1$, or the end of the pipeline is reached. If the boundary falls inside an indivisible stage, move it back to the beginning of the stage. Finally, recursively execute this algorithm on the remainder of the pipeline and the remaining resources in the chain. If all work in the pipeline can be mapped to at most n resources with this algorithm, return "true"; else return "false". Pseudocode for this procedure, called VERIFYPERIOD, is given in Algorithm 2.

Claim V.1. *Algorithm 2 correctly determines whether the pipeline can be mapped to the given resources with period at most τ.*

Proof: The algorithm never allocates more than $\tau \cdot sp_j$ normalized work to the jth resource. Hence, every resource is assigned scaled work at most τ, and so, if the pipeline is completely mapped (i.e. the algorithm returns "true"), the period of the mapping is at most τ.

Conversely, suppose that there exists a mapping M with period at most τ. We will show that M can be transformed into the mapping found by the greedy algorithm while maintaining a period of at most τ. We proceed by induction on the number of resources n.

Bas: if $n = 1$, then M assigns all stages to resource P_1; hence, the greedy algorithm can also assign all stages to this resource while achieving the same period $\leq \tau$.
Ind: The mapping M assigns normalized work at most $\tau \cdot sp_1$ to resource P_1. If it assigns exactly this much work, or the boundary of P_1's component is at the start of an indivisible stage that, if added, would cause the P_1's normalized work to exceed $\tau \cdot sp_1$, then the component is the same as that assigned by the greedy algorithm. Otherwise, M assigns strictly less work to P_1 than the greedy algorithm, and we may move work from later components back to P_1's component until it matches the greedy algorithm's result. Now consider the portion of the pipeline not mapped to P_1, and let M' be the induced mapping of this remainder onto resources $P_2 \dots P_n$ after the above transformation. Clearly, M' also has period at most τ; hence, by the inductive hypothesis, we can reallocate its work among $P_2 \dots P_n$ to match the greedy algorithm's result.

Conclude that for any number of resources, M can be transformed to the greedy mapping while maintaining period at most τ, and so the algorithm will return "true". ■

Finally, we analyze the time complexity of this approximation algorithm. Each invocation of VERIFYPERIOD for a given τ runs in time $O(m+n)$. The number of invocations needed for the exponential phase of the search is $O(\log(\frac{\tau^*}{\hat{\tau}}))$, while the number needed for the binary phase is
$$O\left(\log\left(\frac{1}{\epsilon} \cdot \frac{\tau^*}{\hat{\tau}}\right)\right).$$
Now the ratio $\tau^*/\hat{\tau}$ is at most n, since there is always a feasible solution that maps all pipeline stages to the single fastest resource, resulting in a period at most n times $\hat{\tau}$. Conclude that the total number of search steps is $O(\log(\frac{1}{\epsilon}) + \log n)$. Hence, for any fixed ϵ, the complexity of the full algorithm is $O((m+n)\log n)$.

VI. THROUGHPUT OPTIMIZATION ON FULLY CONNECTED HETEROGENEOUS PLATFORMS

In this section, we address the problem of mapping partially replicable pipelines onto heterogeneous cliques. As mentioned in Section I, mapping even a non-replicable pipeline onto a fully connected heterogeneous clique is NP-complete [1], [3]. In this section, we therefore turn to heuristic algorithms. We first explain the ideas behind our heuristics, then present some empirical results demonstrating their effectiveness.

Heuristics for Heterogeneous Cliques

A clique with n distinct resources has $n!$ permutations of unidirectional resource chains. For each chain, we can use the dynamic programming algorithm of

Algorithm 2: VerifyPeriod

Input: Pipeline, resources, target period
Output: True or False

Function: VERIFYPERIOD(Π, $P_1 \ldots P_n$, τ)
/* Π is a pipeline of stages */

if n = *0* **then**
  if Π *is empty* **then**
    return True
  else
    return False
else
  set right boundary b of component in Π to assign normalized work $\tau \cdot sp_1$, or all remaining work if less, to P_1.
  if b *lies inside an indivisible stage* **then**
    move b to the left boundary of this stage
  let Π' be the unmapped remainder of pipeline
  VERIFYPERIOD(Π', $P_2 \ldots P_n$, τ)

Section V to compute an optimal mapping in polynomial time, but finding the global optimum requires $n!$ such calls. We attempted to exhaustively optimize over $8!$ chains for a 40-stage pipeline and found that the computation required six days on a 2.2-GHz AMD Opteron processor. This brute-force approach scales poorly and may be impractical for mapping onto large numbers of resources. We therefore devise heuristics to find good, if not optimal, mappings. We consider three heuristic approaches: *random sampling*, *hill climbing*, and *simulated annealing*.

In the random sampling heuristic, we sample random resource chains from the clique and compute an optimal mapping of the pipeline to each chain, keeping the best mapping found. This is the simplest strategy.

In hill climbing, we start with an arbitrary resource chain, then incrementally improve the solution by generating a new chain from the previous one. We call this process ***bottleneck alleviation***. Suppose we have a mapping in which P_i has speed sp_i and is assigned work $w(P_i)$. Say the period of the mapping is $\tau = w(P_i)/\, sp_i$ (that is, P_i is a bottleneck). We try to find another resource P_j which can alleviate this bottleneck. This happens if P_j with speed sp_j and assigned work $w(P_j)$ satisfies $w(P_i)/\, sp_j < \tau$ and $w(P_j)/\, sp_i < \tau$. In this case, we can swap P_i and P_j in the chain and obtain a new chain. If P_i was the only bottleneck resource in the old solution, then we have strictly reduced the period. In any cases, the period does not increase. We may be able to further improve the period by computing an optimal mapping onto the new chain.

After a few steps, hill climbing typically reaches a local optimum for which no resource swap leads to a better solution. To continue exploring chains, we must restart the climbing process from some other chain. We tried several restarting policies, including restarting from a random chain and restarting from a neighboring chain, but found no significant variation in performance among them. We finally adopted an ***adaptive restarting*** policy, in which we randomly choose a chain whose distance (in terms of number of resource swaps) from the initial chain grows as we explore more chains.

Simulated annealing [18] is a generic method that mimics how atoms in a heated metal adjust their population during the cooling process. It explores the solution space by iteratively attempting to move from the current solution to a neighboring one. If the neighbor is better, the search will move to it; if the neighbor is worse, the search will still move to it with a probability determined by the current "temperature" and the objective value difference between the two solutions. The higher the temperature, the more likely the move. The temperature gradually drops until, when it reaches zero, the algorithm behaves equivalently to hill-climbing.

Experimental Setup

In Section V, we described two algorithms to map a pipeline onto a heterogeneous chain: an exact algorithm with time complexity $O(m^3n^3)$, and a $(1+\epsilon)$-approximation algorithm. Note that we must use one of these chain algorithms as a subroutine for both our brute-force search and heuristic search. On our hardware, using the exact method proved impractical even for the heuristic search. Therefore, we use the approximation algorithm with $\epsilon = 0.01$ for all the following results.

We generated three test cases: test case 1 with 20 stages and 8 resources (7 divisible, 13 not divisible), test case 2 with 50 stages and 16 resources (13 divisible, 37 not divisible), and test case 3 with 100 stages and 32 resources (28 divisible, 72 not divisible). Although it is impossible to cover all application scenarios, the three test cases demonstrate the effectiveness of our algorithms for reasonably large inputs. For each test case, we wanted to compute the optimal mapping using a brute force search in order to quantify the results from the heuristics. For the 20-stage, 8-resource test case, we simply used brute-force enumeration of chains together with the approximation algorithm with $\epsilon = 0.01$. However, solving the 16-resource test case would take an estimated 241 CPU-years by this method. For the larger test cases, we bypassed this huge computational cost by changing a few stages from "stateful" to "stateless", so that a perfectly divided mapping, which is guaranteed to be optimal, was known *a priori* to exist for some resource chain.

Experimental Results

Table I compares the mean periods ($\pm$ one standard deviation) achieved by different methods and the time spent in computing for each test case. BF, RS, HC, and SA stand for the brute-force search, random sampling, hill climbing, and simulated annealing, respectively. We do not have timing results for brute force search for the larger pipelines, since it would have taken too long, and we simply tweaked the pipelines to be perfectly divisible. We ran each heuristic with a fixed number of iterations, where each iteration explores one chain. The iteration count was 2,000 for test case 1, 20,000 for test case 2, and 100,000 for test case 3. Beyond these iteration counts, the three heuristics perform similarly, and performances are improved very slowly.

Figures 4, 5, and 6 illustrate the performance of the heuristics on the three test cases with different numbers of iterations. The number of iterations is plotted on the x-axis, and the best period found is on the y-axis. We repeated each computation 100 times with different random seeds to estimate the variability in each heuristic's performance. The curves show the average period values found by the heuristics after each number of iterations. Whiskers on the data points represent the standard deviation. (We avoid making claims of error bars since we have no reason to expect the distribution to be Gaussian.) The horizontal line in each figure is the near-optimal period for the test case (determined using brute force search with the approximation chain algorithm for the smallest test case, and determined using the PDM algorithm for the larger test cases).

The three heuristics all performed similarly, finding mean solutions that achieve 1.05$\times$ of the optimal period within 2,000 iterations. For the smallest test case, they all found the optimal solution. Random sampling performed slightly better in the mean than the other two; however, the variation between the techniques is substantially smaller than the variation within each technique (i.e., the separation between the means is smaller than their individual standard deviations).

VII. Conclusion

We have presented algorithms for mapping partially replicable pipelines onto homogeneous and heterogeneous platforms. We show that polynomial-time dynamic programming algorithms can yield optimal results for homogeneous platforms and for heterogeneous chains. For heterogeneous cliques, where no polynomial-time algorithm likely exists, we have presented heuristics, and our empirical results show that these heuristics find near-optimal solutions quickly. Our algorithms can *arbitrarily* divide replicable tasks to achieve optimal or near-optimal throughput, which previous algorithms cannot do.

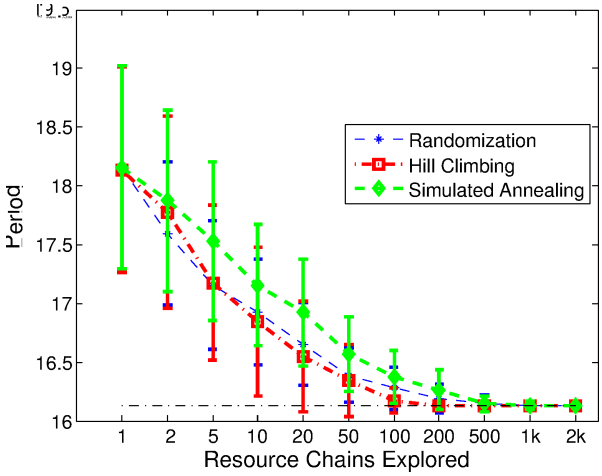

Figure 4. Test case 1: 20 stages and 8 resources.

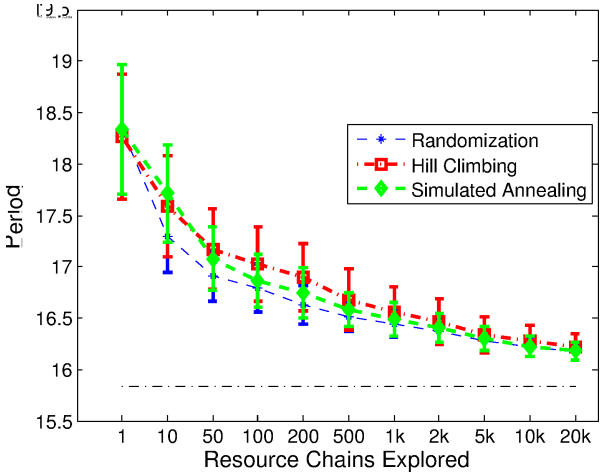

Figure 5. Test case 2: 50 stages and 16 resources.

There are many directions for future work. First, we would like to incorporate communication costs into the model while deciding the mappings. Second, we want to reduce the complexity of the dynamic programming-based algorithms proposed in this paper. Third, even though the heuristics of Section VI perform well in practice, they provide no theoretical guarantees. We would like to design approximation algorithms for mapping onto heterogeneous cliques.

References

[1] A. Benoit and Y. Robert, "Mapping pipeline skeletons onto heterogeneous platforms," *J. Parallel Distrib. Comput.*, vol. 68, no. 6, pp. 790–808, Jun. 2008.

[2] K. Agrawal, A. Benoit, and Y. Robert, "Mapping linear workflows with computation/communication overlap," in *Proc. of 14th IEEE Int'l Conf. on Parallel and Distributed Systems*, 2008, pp. 195–202.

Table I
AN OVERVIEW OF RESULTS.

	Test Case 1		Test Case 2		Test Case 3	
	Period	Time	Period	Time	Period	Time
BF	16.13	9.22s	15.83	N/A	16.47	N/A
RS	$16.13 \pm 2 \times 10^{-14}$	0.47s	16.17 ± 0.08	9.01s	17.07 ± 0.08	85.79s
HC	$16.13 \pm 2 \times 10^{-14}$	0.52s	16.21 ± 0.13	9.33s	17.08 ± 0.16	86.44s
SA	$16.13 \pm 2 \times 10^{-14}$	0.47s	16.18 ± 0.09	9.00s	17.08 ± 0.08	85.06s

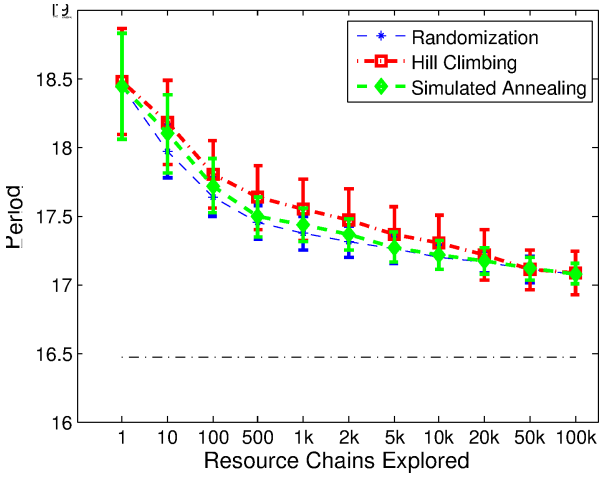

Figure 6. Test case 3: 100 stages and 32 resources.

[3] A. Pınar and C. Aykanat, "Fast optimal load balancing algorithms for 1d partitioning," *Journal of Parallel and Distributed Computing*, vol. 64, no. 8, 2004.

[4] S. H. Bokhari, "Partitioning problems in parallel, pipeline, and distributed computing," *IEEE Trans. on Computers*, vol. 37, no. 1, pp. 48–57, Jan. 1988.

[5] P. Hansen and K.-W. Lih, "Improved algorithms for partitioning problems in parallel, pipelined, and distributed computing," *IEEE Trans. on Computers*, vol. 41, no. 6, pp. 769 –771, Jun. 1992.

[6] H.-A. Choi and B. Narahari, "Algorithms for mapping and partitioning chain structured parallel computations," in *ICPP (1)*, 1991, pp. 625–628.

[7] B. Olstad and F. Manne, "Efficient partitioning of sequences," *Computers, IEEE Transactions on*, vol. 44, no. 11, pp. 1322–1326, 1995.

[8] K. Agrawal, A. Benoit, F. Dufoss, and Y. Robert, "Mapping filtering streaming applications," *Algorithmica*, vol. 62, pp. 258–308, 2012.

[9] J. Subhlok and G. Vondran, "Optimal mapping of sequences of data parallel tasks," in *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 1995, pp. 134–143.

[10] ——, "Optimal latency-throughput tradeoffs for data parallel pipelines," in *Proc. of 8th ACM Symp. on Parallel Algorithms and Architectures*, 1996, pp. 62–71.

[11] M. Kudlur and S. Mahlke, "Orchestrating the execution of stream programs on multicore platforms," in *Proc. of ACM SIGPLAN Conf. on Programming Language Design and Implementation*, 2008, pp. 114–124.

[12] D. Cordes, A. Heinig, P. Marwedel, and A. Mallik, "Automatic extraction of pipeline parallelism for embedded software using linear programming," in *Proc. of IEEE 17th Int'l Conf. on Parallel and Distributed Systems*, 2011, pp. 699–706.

[13] M. I. Gordon, W. Thies, and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," in *Proc. of 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, 2006, pp. 151–162.

[14] Z. Wang and M. F. O'Boyle, "Partitioning streaming parallelism for multi-cores: a machine learning based approach," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 307–318.

[15] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, Sep. 1987.

[16] K. Agrawal, J. Fineman, J. Krage, C. Leiserson, and S. Toledo, "Cache-conscious scheduling of streaming applications," in *Proc. of ACM Symposium on Parallelism in Algorithms and Architectures*, 2012.

[17] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems: theoretical and practical results," *Journal of the ACM (JACM)*, vol. 34, no. 1, pp. 144–162, 1987.

[18] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 222, no. 4598, pp. 671–680, May 1983.