Performance Tuning of Streaming Applications via Search-space Decomposition

Authors: Shobana Padmanabhan, Roger D. Chamberlain, and Yixin Chen

High-performance streaming applications are typically pipelined and deployed on architecturally diverse (hybrid) systems. Developers of such applications are interested in customizing components used, so as to benefit application performance. We present an efficient and automatic technique for design-space exploration of applications in this problem domain. We solve performance tuning as an optimization problem by formulating cost functions using results from queueing theory. This results in a mixed-integer nonlinear optimization problem which is NP-hard. We reduce the search complexity by decomposing the search space. We have developed a domain-specific decomposition technique using topological information of the application embodied in the queueing network models. Our analysis includes when our decomposition preserves optimality. Our preliminary empirical results confirm two-fold benefits--solving a problem that is currently not solvable using state-of-the-art solvers and in some problem instances, improving initial solution value from the solver by over two orders of magnitude.

... Read complete abstract on page 2.

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
https://openscholarship.wustl.edu/cse_research/37
Performance Tuning of Streaming Applications via Search-space Decomposition

Complete Abstract:

High-performance streaming applications are typically pipelined and deployed on architecturally diverse (hybrid) systems. Developers of such applications are interested in customizing components used, so as to benefit application performance. We present an efficient and automatic technique for design-space exploration of applications in this problem domain. We solve performance tuning as an optimization problem by formulating cost functions using results from queueing theory. This results in a mixed-integer nonlinear optimization problem which is NP-hard. We reduce the search complexity by decomposing the search space. We have developed a domain-specific decomposition technique using topological information of the application embodied in the queueing network models. Our analysis includes when our decomposition preserves optimality. Our preliminary empirical results confirm two-fold benefits—solving a problem that is currently not solvable using state-of-the-art solvers and in some problem instances, improving initial solution value from the solver by over two orders of magnitude.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/37
Performance Tuning of Streaming Applications via Search-space Decomposition

Authors: Shobana Padmanabhan, Roger D. Chamberlain, Yixin Chen

Corresponding Author: shobana@arl.wustl.edu

Abstract: High-performance streaming applications are typically pipelined and deployed on architecturally diverse (hybrid) systems. Developers of such applications are interested in customizing components used, so as to benefit application performance. We present an efficient and automatic technique for design-space exploration of applications in this problem domain.

We solve performance tuning as an optimization problem by formulating cost functions using results from queueing theory. This results in a mixed-integer nonlinear optimization problem which is NP-hard. We reduce the search complexity by decomposing the search space. We have developed a domain-specific decomposition technique using topological information of the application embodied in the queueing network models. Our analysis includes when our decomposition preserves optimality. Our preliminary empirical results confirm two-fold benefits—solving a problem that is currently not solvable using state-of-the-art solvers and in some problem instances, improving initial solution value from the solver by over two orders of magnitude.

Type of Report: Other
Performance Tuning of Streaming Applications via Search-space Decomposition

Shobana Padmanabhan, Roger D. Chamberlain, and Yixin Chen
Dept. of Computer Science and Engineering, Washington University in St. Louis

Abstract

High-performance streaming applications are typically pipelined and deployed on architecturally diverse (hybrid) systems. Developers of such applications are interested in customizing components used, so as to benefit application performance. We present an efficient and automatic technique for design-space exploration of applications in this problem domain.

We solve performance tuning as an optimization problem by formulating cost functions using results from queueing theory. This results in a mixed-integer nonlinear optimization problem which is NP-hard. We reduce the search complexity by decomposing the search space. We have developed a domain-specific decomposition technique using topological information of the application embodied in the queueing network models. Our analysis includes when our decomposition preserves optimality. Our preliminary empirical results confirm two-fold benefits—solving a problem that is currently not solvable using state-of-the-art solvers and in some problem instances, improving initial solution value from the solver by over two orders of magnitude.

1 Introduction

In streaming applications, application data is fed forward over a pipeline of processing elements to achieve high performance. Examples of streaming applications abound in the areas of biosequence analysis, computer networking, signal processing, video processing, image processing, and computational science. Streaming applications are sufficiently widespread that several programming languages have been developed for them, including Brook [4], StreamIt [19], and X [11].

We define performance tuning as finding the best values for the different performance-related design parameters, given performance goals and constraints. Examples of design parameters are sizes of buffers in the processing elements, delivery size of messages communicated between processing elements, mapping of processing and communication elements onto physical resources and choices of algorithms. Performance tuning is hard because:

- The number of possible configurations is exponential in the number of performance parameters. For our example streaming sort application there are over $10^{21}$ possible configurations.
- Performance goals are often multiple and conflicting. For instance, it is typical to minimize application latency (end-to-end execution time) while simultaneously maximizing application throughput (input data rate), but these goals often conflict with each other.

Performance tuning has been an active research topic in numerous application areas. For embedded applications, performance tuning has been researched using search heuristics as well as standard and modified optimization techniques [9, 10, 17]. A recent trend has been to model the application’s performance analytically (mathematically) and use the model with search heuristics. Examples of such models include predictive models [12, 14] and examples of search heuristics include gradient ascent [14]. Predictive models are based on regression or machine learning and trained with empirical experimentation through simulation or direct execution. Such models are general and hence can be applied to any design space.

Our approach is to solve performance tuning as an optimization problem; for modeling cost functions, we prefer queueing networks (QNs) over regression-based models. This is because pipelining in the streaming applications gives the application special structure and frequently involves queueing between the processing elements. Examples of streaming applications that have been modeled using queueing networks include computer systems, computer networking, software architecture, biosequence search, and web servers [5, 13, 21].

The resulting optimization problem formulation is a mixed-integer nonlinear problem (MINLP). A characteristic of MINLP problems is that the search time is known to grow exponentially with respect to the number of variables [3]. Further, the nonlinear functions in our optimization problem are neither convex nor quasiconvex which means there is no theory that guarantees finding a global optimum. In practice, state-of-the-art solvers failed to find even a feasible solution to our problem.
A popular approach with MINLP problems is to decompose the search space in order to obtain a better solution from optimization [8]. However, the state-of-the-art solvers [16] such as Bonmin, FilMINT, KNITRO, and MINLP, fail to even find a feasible solution for our problem formulation. It is a common practice to use domain-specific information to guide decomposition [20]. However, no such technique exists for our problem domain.

We have developed a domain-specific decomposition technique that:

- Identifies and characterizes topological information in the domain and uses it to guide the decomposition.
- Identifies the specific MINLP form of problems in the domain.
- Results in a heuristic to order variables and constraints so as to improve decomposability and handle complications. Our analysis includes when our decomposition preserves optimality.

We describe a preliminary empirical validation of the benefits of our techniques. The two-fold benefits are: we solve a problem that is currently not solvable using state-of-the-art solvers, and for a relaxed version of the problem that is solvable, we improve the initial solution value by over two orders of magnitude.

2 Domain-specific decomposition

In general, a problem is considered decomposable if we can arrange the variables and constraints such that a matrix showing the presence of the variables in the different constraints has the canonical Jordan block form [18]. This form is illustrated in Figure 1. In this form, the objective function also decomposes to correspond to the different blocks (i.e., we can solve $Obj_1$ along with the variables and constraints forming $block_1$ as an independent subproblem). Note that decomposing this way preserves optimality.

A variable that prevents decomposition into blocks is traditionally referred to as a “complicating variable” (CV) and a constraint that prevents decomposition is called a “complicating constraint” (CC). The effects of these complications on the Jordan canonical form are shown in Figure 2 and Figure 3 respectively. Decomposition techniques exist for dealing with CVs and CCs [8]. For MINLP problems in particular, integer variables are treated as CVs and if the resulting nonlinear program (NLP) is convex at least locally, Benders decomposition is known to converge to an optimal solution (or to some small duality gap). Another approach with MINLP problems is to consider the nonlinear constraints as CCs and apply the outer linearization algorithm, provided the nonlinear constraints are inequalities and the objective function is linear (both of which can be usually achieved through simple transformations).

In real-world streaming applications however, the number of CVs and CCs tend to exceed the number of non-complicating variables and constraints making it very hard to achieve decomposability. Hence, we have developed a domain-specific decomposition technique that exploits the pipelining structure of the streaming applications.

**Domain-specific topological information** We characterize the topological information about streaming applications modeled by queueing networks (QNs) as follows.

- $ip = \{ ip_i | ip_i \in \mathbb{R}_+ \cup \{0\} \}$ is the set of input parameters whose optimal settings are to be determined. In practice, most of the parameters are integer- or binary-
• \( \text{var} = \{ \text{var} \mid \text{var} \in \mathbb{R}^+ \cup \{0\} \} \) is the set of design variables in the optimization problem formulation that correspond to the input parameters. \( n_v = |\text{var}| \) and \( v \in (\mathbb{R}^+ \cup \{0\})^{n_v} \) is the vector formed by the variables in \( \text{var} \).

In most cases, the design variables are the same as the input parameters. An example of design variables that are distinct from their corresponding input parameters arises in the following context. Let \( m \in \mathbb{Z}^+ \) be an input parameter that controls mapping choices. In our optimization model, we transform \( m \) into a set of binary design variables, one for each mapping choice, and constrain them so that only one of them is selected.

• \( \text{top} \subseteq \text{var} \) is the set of design variables that result in distinct alternative QNs. \( n_t = |\text{top}| \) and \( t \) is the vector formed by the variables in \( \text{top} \). We call these variables topological variables, since they impact the topology of the queueing network.

• \( \text{der} = \{ \text{der} \mid \text{der} \in (\mathbb{R}^+ \cup \{0\}) \} \) is the set of derived variables that depend on one or more elements in \( \text{var} \). \( n_d = |\text{der}| \) and \( \text{d} \) is the vector formed by the variables in \( \text{der} \).

• \( \text{met} \subseteq \text{der} \) is the set of performance metrics (such as application latency, throughput, power consumption, etc.) that are being optimized. \( n_m = |\text{met}| \) and \( \text{met} \) is the vector formed by the variables in \( \text{met} \).

• \( z \in \text{der} \) is the cost function, also known as the objective function.

• \( \text{ivar} \subseteq \text{der} \) is the set of intermediary variables (IVs). \( n_q = |\text{ivar}| \) and \( q \) is the vector formed by the variables in \( \text{ivar} \). \( \text{ivar} = \text{der} \setminus \text{met} \) \(-\) \( \{z\} \).

Intermediate variables may arise for a number of reasons: (1) application developers may be interested in the values of IVs for debugging purposes; (2) IVs can codify abstractions in the performance models such as QNs; and (3) IVs may help some search heuristics. For example, MINLP solvers using cutting-plane based algorithms, such as FilMINT [2], tend to work better with linear cost functions. While solvers using the interior-point algorithm tend to work better with linear constraints [3].

• \( \text{mu} \subseteq \text{ivar} \) is the set of mean service rates at each queueing station. \( n_u = |\text{mu}| \) and \( \mu \) is the vector formed by the variables in \( \text{mu} \).

\[
\mu_j = u_j(v) : (\mathbb{Z}^+ \cup \{0\})^{n_v} \rightarrow \mathbb{R}^+ \text{ where } u \text{ is the vector of functions that define } \mu.
\]

• \( \text{lam} \subseteq \text{ivar} \) is the set of mean job arrival rates at each queueing station. \( n_l = |\text{lam}| \) and \( \lambda \) is the vector formed by the variables in \( \text{lam} \). \( \lambda \) is related by a system of linear expressions with a unique solution (in terms of the input parameter \( \lambda_{in} \in \text{var} \)).

\[
\lambda_j = l_j(\lambda_{in}, t) : (\mathbb{R}_+ \cup \{0\})^{n_v} \rightarrow (\mathbb{R}_+ \cup \{0\}) \text{ where } l \text{ is the vector of functions that define } \lambda.
\]

• \( \text{sq} \subseteq \text{var} \) is the set of Single-QS (SQ) variables. Each element of \( \text{sq} \) is in the domain of only one \( u_j(\cdot) \). \( n_s = |\text{sq}| \) and \( s \) is the vector formed by the variables in \( \text{sq} \).

• \( \text{mq} \subseteq \text{var} \) is the set of Multi-QS (MQ) variables. Each element of \( \text{mq} \) is in the domain of more than one \( u_j(\cdot) \). \( n_m = |\text{mq}| \) and \( m \) is the vector formed by the variables in \( \text{mq} \).

• \( \text{ul} \) is a vector of model constraints that restricts every \( \lambda_j < \mu_j \) for the system to be stable.

Topologies of streaming applications are restricted to directed acyclic graphs while QN topologies are annotated digraphs. Annotations on each digraph include, at the minimum, expressions for each of \( \mu, \lambda, \) and \( o \). Each node of the digraph represents a queueing station (QS) which is a service facility with its queue. Each edge represents the communication link between two nodes.

Domain-specific MINLP form The general form of the optimization function of a streaming application modeled using queueing networks is as follows. The nonlinear functions may not be continuous or differentiable.

\[
\begin{align*}
\text{min} & \quad \sum_{k=1}^{n_m} W_k \times \text{met}_k : (\mathbb{R}_+ \cup \{0\})^{n_v} \rightarrow \mathbb{R}_+, \sum_{k=1}^{n} W_k = 1 \\
\text{subject to} & \quad u(v) = 0 \\
& \quad l(\lambda_{in}, t) = 0 \\
& \quad \text{ul} = 0
\end{align*}
\]

The presence of \( t \) is a discovery from our experimentation described in Section 4. There may be multiple QNs for a given application topology. If \( \text{top} \neq \emptyset \), we use a set of binary variables to model the QNs resulting from every variable in \( t \). In that case, there exist (equality) constraints to choose only one of the binary variables from each set.

Domain-specific complicating variables and optimality It is our observation that the topological variables \( t \) are special complicating variables in that decomposing by them results in completely independent subproblems by definition. That means resolving solutions from the subproblems
is trivial because we only need to identify the minimum valued solution. That also means decomposing by these variables will preserve optimality.

On the other hand, λ and multi-QS variables (m) are complicating variables that will have to be resolved if decomposed over. Because λ are constrained by μ through ul, μ have the same effect on decomposition as the λ. In our ongoing work, we handle this case by implementing branch-and-bound where we use the variable-constraint ordering described below to determine the branching order.

**Domain-specific variable-constraint ordering** To identify decomposability, we have developed the following heuristic to order the variables and constraints our variable-constraint matrix gets closer to the canonical Jordan block form as mentioned at the beginning of this section.

1. Start with sq. If multiple elements in sq, include them in the order of the pipeline which is also the order of the queueing stations in the QN model. Include first all the variables from sq that concern a given QS. For example, include all SQ variables that affect QS 0 and then include all SQ variables that affect QS 1 etc.

2. Include MQ variables relating, increasingly, to one or more QS variables. When relating to multiple other QS variables, they are ordered in the direction of the application’s pipeline.

3. Include variables from top, i.var, met, and finally z.

4. Order the constraints, starting with u and then l. Include the constraints, again, in the direction of the pipeline. For example, include the constraints that are isolated to QS 0 before including those for QS 1 and so on. If all the constraints in u and l span QSs, then, include them so that the constraints for relating fewer QS variables are before the ones relating more QS variables.

5. Follow with the equality and inequality constraints of ul, o, and then the binary constraints for t.

We use the ordering of variables and constraints to guide our decomposition of the search space. For the complicating variables that are elements of top, our decomposition works as follows.

1. Start with the CV that appears in the most number of constraints and evaluate each of its values as an independent subproblem (i.e., branch on the CV).

2. For each subproblem at this step, we decompose the subproblem by the next CV.

3. Repeat until we have decomposed all the CVs.

By the nature of applications in our problem domain, we do not expect all CVs to be from top. Dealing with CVs not in top is our ongoing work. We are implementing a branch-and-bound algorithm guided by the ordering of variables and constraints discussed above.

### 3 Example Application

Our example application is a streaming (parallel) sort. Although sorting is a simple application, it is prototypical of how streaming applications get parallelized for high performance. In addition, the number and type of performance-related parameters for streaming sort on our deployment platform is also prototypical.

In streaming sort, input data is split into parts and each part is sent to a sort instantiation. This application topology is shown in Figure 4. The sort instantiation is referred to as a sort “block” in the parlance of Auto-Pipe [11]. The sort blocks execute in parallel and when done, each block sends its output to a merge block. The merge block then merges its inputs and sends out the sorted data. The edges between the blocks are communication links. A “column" refers to all the blocks or links at the same level in Figure 4.

The set of design variables, var, along with their characterization, bounds and constraints, for streaming sort are enumerated in Figure 5. Note that the number of sort blocks, denoted by $2^N$, controls the degree of parallelization which in turn controls the tradeoff between application latency and throughput, reflected in a change in the application topology as illustrated in Figure 6.

Mapping of the application blocks and edges onto available physical resources is an important concern during application deployment. Here, we consider only a few interesting and prototypical mappings. The choices are represented by the binary variables $m_0, m_1, m_{CHR}, m_{HHR}$ which are constrained to have only one of them be true. $m_0$ models
when every block in the application gets its own set of resources, \( m_1 \) models when all compute blocks share a single resource, \( m_{CB} \) models when all split blocks share a single computation resource, and \( m_{IR} \) models when the communication links into and out of the sort blocks are shared. The variables controlling both the number of sort blocks and the mapping are topology variables, elements of \( \text{top} \).

**Queueing network model** Queueing network models represent a system as an interconnected set of queueing stations and customers (jobs) serviced by those queueing stations. Each queueing station has one or more servers. Queueing stations are conventionally labeled with the notation \( a/b/s \), where \( a \) and \( b \) represent the distribution of interarrival and service times respectively; and \( s \) represents the number of servers. The model we use initially is the classic \( M/M/1 \) model where \( M \) is an exponential (memoryless or Markovian) distribution.

If in a BCMP queueing network each station has an infinite queue, it follows from the equivalence property that (under steady-state conditions) each station can be analyzed independently [2]. Here, we restrict ourselves to \( M/M/1 \) BCMP networks with infinite buffers and FIFO queuing discipline. We make these assumptions only to prevent complicating the analytical expressions. Inherently, our work can handle relaxation of each of these assumptions as long as there are known results in queueing theory to handle the relaxation. See, e.g., [13] for approaches to handling finite queue sizes and/or phase-type service distributions.

For the sort application, we begin with the application topology shown in Figure 4. The queueing network model for this topology is shown in Figure 7. Note that we model each *column* as an individual queueing station. Some mapping choices change the queueing network’s topology. An example of such a mapping choice is \( m_{IR} \). The resulting queueing network in illustrated in Figure 8, where the server “Comm” is handling all of the communication both into and out of the “Sort” server.

**Cost functions and validation** Recall from Section 2 that expressions for \( \mu \) are given by \( u \) and are generally input by the application developer. For our example application, we derived these expressions based on first principles and

<table>
<thead>
<tr>
<th>Variable</th>
<th>Symbol</th>
<th>Ranges and Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of elements to be sorted</td>
<td>( 2^M )</td>
<td>( B_{SZ} = K, K \in \mathbb{Z}_+ ) (e.g., 20).</td>
</tr>
<tr>
<td>Number of sort blocks</td>
<td>( 2^N )</td>
<td>( N = 1, 2, 3, \ldots, 2^M; N \in \top )</td>
</tr>
<tr>
<td>Index of split columns</td>
<td>( j )</td>
<td>( j = 0, 2, \ldots, 2N - 1 )</td>
</tr>
<tr>
<td>of link columns left of sort column of sort column</td>
<td>( j )</td>
<td>( j = 1, 3, \ldots, 2N - 1 )</td>
</tr>
<tr>
<td>of link columns right of sort column of merge columns</td>
<td>( j )</td>
<td>( j = 2N )</td>
</tr>
<tr>
<td>of link columns right of sort column of merge columns</td>
<td>( j )</td>
<td>( j = 2N + 1, 2N + 3, \ldots, 4N - 1 )</td>
</tr>
<tr>
<td>Communication resource type</td>
<td>binary: ( \text{cpu}_j ) or ( \text{fpga}_j )</td>
<td>( j = 0, 2, \ldots, 4N; \text{cpu}_j + \text{fpga}_j = 1 )</td>
</tr>
<tr>
<td>Communication resource type</td>
<td>binary: ( \text{smem}_j ) or ( \text{gige}_j )</td>
<td>( j = 1, 3, \ldots, 4N - 1; \text{smem}_j + \text{gige}_j = 1 )</td>
</tr>
<tr>
<td>Number of compute resources</td>
<td>( n\text{Res}_j )</td>
<td>( \forall ) split columns: ( n\text{Res}_j \leq 2^j ); ( \forall ) sort columns: ( n\text{Res}_j \leq 2^j )</td>
</tr>
<tr>
<td>Number of communication resources</td>
<td>( n\text{Res}_j )</td>
<td>( \forall ) merge columns: ( n\text{Res}_j \leq 2^{j-1} )</td>
</tr>
<tr>
<td>Mapping choices</td>
<td>binary: ( m_0, m_1, m_{CB}, m_{IR} )</td>
<td>( m_0 + m_1 + m_{CB} + m_{IR} = 1 ); each ( m_i \in \text{top} )</td>
</tr>
<tr>
<td>System-wide comm message size</td>
<td>( 2^M )</td>
<td>( M = 0, 1, \ldots, M_{IB} ) where ( M_{IB} \leq N ) and ( M_{IB} = K, K \in \mathbb{Z}_+ ) (e.g., 14)</td>
</tr>
<tr>
<td>Sort algorithm (only with ( \text{cpu}_j ) mapping)</td>
<td>binary: ( \text{alg1} ) or ( \text{alg2} )</td>
<td>( m_1(\text{fpga}[0] + \text{alg1} + \text{alg2}) + (1 - m_1)(\text{fpga}[j] + \text{alg1} + \text{alg2}) = 1, j = 2N )</td>
</tr>
<tr>
<td>Input mean job arrival rate</td>
<td>( \lambda_{in} \in \mathbb{R}_+ )</td>
<td>By solving ( u )</td>
</tr>
</tbody>
</table>
Formulate optimization problem We use the standard weighted sum technique to combine the multiple (normalized) performance objectives (met from Section 2) as shown in equation 4. Note that if we optimized only the application’s throughput, given by \( \frac{1}{\mu} \), the problem degenerates to identifying the bottleneck in the pipeline.

\[
\text{minimize} \quad W_1 \times \text{Latency} + W_2 \times \frac{1}{\lambda_{in}}, \sum_{i=1}^{2} W_i = 1 \tag{4}
\]

We use the equivalence property of M/M/1 BCMP networks with infinite buffers and a FIFO queuing discipline to define the mets. Accordingly, latency is given by:

\[
\text{Latency} = \sum_{j=0}^{4N} \frac{1}{\mu_j - \lambda_j} \tag{5}
\]

As the example equations above illustrate, the equations are highly nonlinear and the state-of-art-solvers we mentioned earlier are unable to find even a feasible, much less optimum, solution. The number of variables and constraints in the original problem range from (50, 30) to (399, 3077) as we increase \( N \) from 1 to 13. This corresponds to \( 2^{N} = 2 \) to 8192 sort blocks, given a batch size of \( 2^{14} \) elements per sort.

Variables other than \( \mu, \lambda, \text{Latency, and Throughput} \) are integer-valued, making the problem mixed-integer. The expression for latency, handling of multiple mapping choices, and handling the number of sort blocks as a variable all make the problem nonlinear, and some of the nonlinear functions are nonconvex and non-quasiconvex.

4 Preliminary empirical results

We first categorize the variables and use the categories to order the variables and constraints to achieve decomposability per our heuristic described in Section 2. The resulting matrix structure is shown in Figure 9. The matrix shows significant decomposable parts but it also shows the presence of many complicating variables and constraints.

We first decompose the complicating variable \( N \) and then \( m \). \( N \) determines the number of sort blocks, and \( m \) is the set of mapping choices. Let \( p \) be the original problem that evaluates all values of \( N \) and all the mapping choices simultaneously. Let \( p_{i} \) denote the subproblem obtained by decomposing \( p \) by fixing \( N \) at \( i \) (which implies the number of sort blocks is \( 2^{i} \)) but considers all the mapping choices. Then, we decompose each \( p_{i} \) based on \( m \). We denote the subproblem of \( p_{i} \) that considers only \( m_0 \) by \( p_{i,1} \), only \( m_1 \) by \( p_{i,2} \), only \( m_{CR} \) by \( p_{i,3} \), and only \( m_{IR} \) by \( p_{i,4} \). The relationship among the subproblems in the hierarchical decomposition is shown in Figure 10.

The objective function values (equally-weighted sum of application latency and job interarrival time to sort a batch of a million 64-bit elements) from the solutions of each of the subproblems, as solved by FilMINT [1], are presented in Figure 11 (up to \( N = 9 \)). Recall that the solution to the original problem is simply the minimum of the solutions from all the subproblems. This minimum value is highlighted.
is no more than 380 and 326, respectively. 
the number of variables and constraints in any subproblem 
problem is progressively less complex. After decomposing, 
problems instead of just one (original) problem, each sub-

7 s and hence the overhead from decomposition is not a con-

Analysis of empirical results 
The solver runtime for the 
different subproblems ranged from 20 ms to approximately 
7 s and hence the overhead from decomposition is not a concern 
for this problem. Although we are solving many sub-
problems instead of just one (original) problem, each sub-
problem is progressively less complex. After decomposing, 
the number of variables and constraints in any subproblem 
is no more than 380 and 326, respectively.

The solution of every subproblem through our heuristic 
is not guaranteed to be necessarily a local optimum because 
the cost functions and some constraints still remain non-
convex in the subproblems. For our example application, 
an example of solution to a subproblem not being a local 

optimum is the subproblem \( p_{3,0} \). The solution has \( M = 4 \) 
but increasing \( M \) to 5 in a neighborhood search lowers the 
objective function value from 299.02 to 298.87 ms. While 
the difference between these two values is actually not truly 
significant from the point of view of the application developer 
(i.e., the performance models are very unlikely to be 
accurate to that many significant digits), it is significant that 
the solver is unable to find even a local optimum.

Our solution is indeed sensitive to the application’s 
performance goals. For instance, rather than optimizing 
equally for both application latency and throughput, if we 
increase the weight on throughput to 0.9 and reduce the 
weight on latency to 0.1, the recommended configuration 
changes from \( (N = 3, m = m_{1R}) \) to \( (N = 4, m = m_{0}) \) 
with a corresponding objective function value of 184.5 ms.

Problem variation 
The problem instance we have considered so far, the communication architecture depends on 
the resource type selected for the processing elements. For example, in Figure 4 if a software implementation is used 
for each of the split and sort blocks, the communication archi-

tecture can be shared memory or Gigabit Ethernet, but if 
one of the ends is mapped instead to an FPGA, the commu-
nication architecture is automatically set to PCI-X.

The formulation for these constraints, however, is highly 
nonlinear and therefore we relaxed these constraints (and 
simultaneously changed a number of the constants) to form 
a variation of the problem that we call “relaxed streaming 
sort.” The solver does manage to solve this relaxed version 
of the problem. Here, using our decomposition heuristic 
improves the initial solution by 480-fold as shown in Fig-
ure 13. In the figure, solution values are denoted by \( s \) 
and solver runtimes are denoted by \( t \). The subproblem \( p_{4,1} \) 
giving the improvement is highlighted.

**Figure 10:** Decomposition of search space of streaming sort application

<table>
<thead>
<tr>
<th>( N )</th>
<th>all ( m )</th>
<th>( m_0 )</th>
<th>( m_1 )</th>
<th>( m_{CR} )</th>
<th>( m_{IR} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>305.3</td>
<td>305.3</td>
<td>1117.2</td>
<td>305.3</td>
<td>305.0</td>
</tr>
<tr>
<td>2</td>
<td>299.1</td>
<td>299.0</td>
<td>1737.8</td>
<td>426.5</td>
<td>299.1</td>
</tr>
<tr>
<td>3</td>
<td>No soln</td>
<td>299.0</td>
<td>2513.1</td>
<td>No soln</td>
<td>298.8</td>
</tr>
<tr>
<td>4</td>
<td>298.9</td>
<td>298.9</td>
<td>3555.8</td>
<td>931.0</td>
<td>298.9</td>
</tr>
<tr>
<td>5</td>
<td>No soln</td>
<td>298.9</td>
<td>3841.4</td>
<td>No soln</td>
<td>298.9</td>
</tr>
<tr>
<td>7</td>
<td>298.9</td>
<td>No soln</td>
<td>4372.9</td>
<td>No soln</td>
<td>No soln</td>
</tr>
<tr>
<td>8</td>
<td>298.9</td>
<td>298.9</td>
<td>4772.9</td>
<td>No soln</td>
<td>No soln</td>
</tr>
<tr>
<td>9</td>
<td>No soln</td>
<td>299.1</td>
<td>5225.9</td>
<td>No soln</td>
<td>298.9</td>
</tr>
</tbody>
</table>

**Figure 11:** Objective function values (in ms) in the subproblems when optimizing application latency and throughput equally

in the figure, and its configuration is shown in Figure 12. 
"No soln" in Figure 11 means the solver failed to converge 
to a feasible solution for that problem instance. Note that 
there is no additional benefit to increasing parallelism be-
yond \( N = 3 \) for our optimization because the split and 
merge blocks become the bottleneck. It is also the reason 
why many solutions are close to the minimum—the minor 
differences are due to variations in queueing delays.

**Figure 12:** Configuration corresponding to the best solution when optimizing application latency and throughput equally \( (N = 3, m = m_{1R}) \)

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value in the solution configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Res. type for each compute column = FPGA</td>
<td></td>
</tr>
<tr>
<td>Computation res. type determines communication res. type</td>
<td></td>
</tr>
<tr>
<td>Number of resources per column (( n_{Res_j}, n_{Res_j} ))</td>
<td></td>
</tr>
<tr>
<td>( n_{Res_0} )</td>
<td>1</td>
</tr>
<tr>
<td>( n_{Res_1}, n_{Res_2} )</td>
<td>2 each</td>
</tr>
<tr>
<td>( n_{Res_3}, n_{Res_4} )</td>
<td>4 each</td>
</tr>
<tr>
<td>( n_{Res_5}, n_{Res_6} )</td>
<td>8 each</td>
</tr>
<tr>
<td>( n_{Res_7} )</td>
<td>0</td>
</tr>
<tr>
<td>( n_{Res_8}, n_{Res_9} )</td>
<td>4 each</td>
</tr>
<tr>
<td>( n_{Res_{10}}, n_{Res_{11}} )</td>
<td>2 each</td>
</tr>
<tr>
<td>( n_{Res_{12}} )</td>
<td>1</td>
</tr>
</tbody>
</table>

\( M = 14 \) (i.e., message size = \( 2^{14} \) Bytes)

Bottleneck mean service rate (num. of 64-bit elems per ms) 
The final merge, \( \mu_1 = 10.986 \)
Input mean job arrival rate (num. of 64-bit elems per ms) 
\( \lambda_{in} = 4.513 \)
5 Conclusions and Future Work

We have presented a heuristic for automatic application-specific performance tuning of streaming applications modeled using queueing networks by developing a domain-specific decomposition technique using topological information embodied in the queueing network models. Our preliminary empirical results show two-fold benefits—solving a problem that is currently not solvable by the state-of-the-art solvers, and for some problem instances improving the initial solution by over two orders of magnitude.

In performing a neighborhood search on the solution of the optimization problem of some problem instances for our example application, we observed that the reported solution was not even a local optimum. This is expected given that some of the functions in our optimization problem are not convex. An easy fix would be to search the neighborhood of the solution reported by the solver. More interesting would be if we can identify topological information that will guide us to partition such that the resulting optimization solution is at least a local optimum. In addition, we are pursuing a wider application set to verify the effectiveness of these ideas across a large domain of problems.

Acknowledgements

We thank the authors and user community of AMPL, FilMINT, and restricted cubic spline package for R, and You Xu of Washington University.

References


