Report Number: WUCSE-2010-12

2010

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.**

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.

Department of Computer Science & Engineering

2010-12

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

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

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

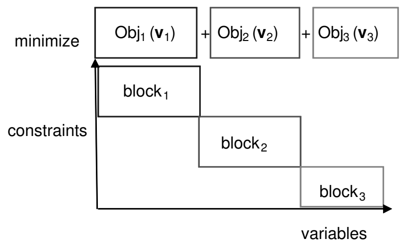

Figure 1: Jordan block form of variable-constraint matrix

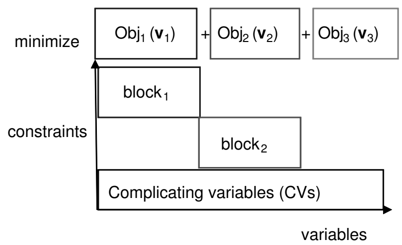

Figure 2: Complicating variables (CVs)

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-

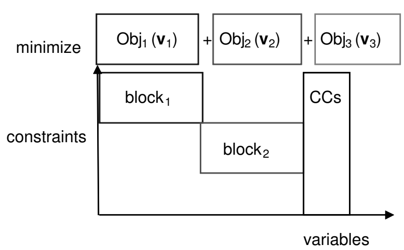

Figure 3: Complicating constraints (CCs)

valued.

- $var = \{var_i | var_i \in \mathbb{R}_+ \cup \{0\}\}$ is the set of **design variables** in the optimization problem formulation that correspond to the input parameters. $n_v = |var|$ and $\mathbf{v} \in (\mathbb{R}_+ \cup \{0\})^{n_v}$ is the vector formed by the variables in 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.

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

- $der = \{der_i | der_i \in \mathbb{R}_+ \cup \{0\}\}$ is the set of **derived variables** that depend on one or more elements in var. $n_d = |der|$ and $\mathbf{d}$ is the vector formed by the variables in der. $der \bigcap var = \emptyset$.

- $met \subseteq der$ is the set of **performance metrics** (such as application latency, throughput, power consumption, etc.) that are being optimized. $n_m = |met|$ and **met** is the vector formed by the variables in *met*.
$met_k = o_k(\mathbf{v}) : (\mathbb{R}_+ \cup \{0\})^{n_v} \longrightarrow \mathbb{R}_+ \cup \{0\}$ where $\mathbf{o}$ is the vector of functions that define **met**.

- $z \in der$ is the cost function, also known as the **objective function**.
$z = \sum_{k=1}^{n_m} W_k \times met_k : (\mathbb{R}_+ \cup \{0\})^{n_m} \longrightarrow \mathbb{R}_+, \sum_{k=1}^{n_m} W_k = 1$

- $ivar \subseteq der$ is the set of **intermediary variables (IVs)**. $n_q = |ivar|$ and $\mathbf{q}$ is the vector formed by the variables in $ivar$. $ivar = der - met - \{z\}$.
Intermediary variables may arise for a number of reasons: (1) application developers maybe 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 [1], tend to work better with linear cost functions while solvers using the interior-point algorithm tend to work better with linear constraints [3].

- $mu \subseteq ivar$ is the set of mean *service* rates at each queueing station. $n_u = |mu|$ and μ is the vector formed by the variables in *mu*.
$\mu_j = u_j(\mathbf{v}) : (\mathbb{Z}_+ \cup \{0\})^{n_v} \longrightarrow \mathbb{R}_+$ where $\mathbf{u}$ is the vector of functions that define μ.

- $lam \subseteq ivar$ is the set of mean *job arrival* rates at each queueing station. $n_l = |lam|$ and λ is the vector formed by the variables in *lam*. λ are related by a system of linear expressions with a unique solution (in terms of the input parameter $\lambda_{in} \in var$).
$\lambda_j = l_j(\lambda_{in}, \mathbf{t}) : \mathbb{R}_+ \cup \{0\} \times \mathbb{Z}_+^{n_t} \longrightarrow \mathbb{R}_+ \cup \{0\}, \lambda_{in} \in \mathbf{v}$ where $\mathbf{l}$ is the vector of functions that define λ.

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

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

- **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 μ, λ, and $\mathbf{o}$. 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.

$$\begin{aligned} \min_{z} \quad & \sum_{1}^{n_m} W_k \times o_k(\mathbf{v}), \sum_{1}^{k} W_k = 1 \\ subject \quad to \quad & \mathbf{u}(\mathbf{v}) = 0 \\ & \mathbf{l}(\lambda_{in}, \mathbf{t}) = 0 \\ & \mathbf{ul}(\lambda, \mu) \leq 0 \end{aligned} \tag{1}$$

The presence of $\mathbf{t}$ is a discovery from our experimentation described in Section 4. There maybe multiple QNs for a given application topology. If $top \neq \emptyset$, we use a set of binary variables to model the QNs resulting from every variable in $\mathbf{t}$. 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 $\mathbf{t}$ 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*, *ivar*, *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.

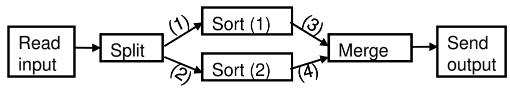

Figure 4: A streaming sort application

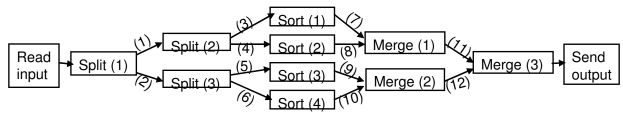

Figure 6: A streaming sort application with 4 sort blocks

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_{CR}, m_{IR} which are constrained to have only one of them be true. m_0 models

Variable	Symbol	Ranges and Constraints
Number of elements to be sorted	2^{B_SZ}	$B\_SZ = K, K \in \mathbb{Z}_+$ (e.g., 20).
Number of sort blocks	2^N	N = 1, 2, 3, ..., $\frac{B\_SZ}{2}$; $N \in top$
Index of split columns of link columns left of sort column of sort column of link columns right of sort column of merge columns	j	$j = 0, 2, ..., 2N - 1$ $j = 1, 3, ..., 2N - 1$ $j = 2N$ $j = 2N + 1, 2N + 3, ..., 4N - 1$ $j = 2N + 2, 2N + 4, ..., 4N$
Compute resource type	binary: cpu_j or $fpga_j$	$j = 0, 2, ..., 4N$; $cpu_j + fpga_j = 1$
Communication resource type	binary: $smem_j$ or $gige_j$	$j = 1, 3, ..., 4N - 1$; $smem_j + gige_j = 1$
Number of compute resources	$nRes_j$	$j = 0, 2, ..., 4N$; $\forall j, nRes_j \geq 1$ $\forall$ split columns: $nRes_j \leq 2^{\frac{j}{2}}$; $\forall$ sort columns: $nRes_j \leq 2^{\frac{j}{2}}$ $\forall$ merge columns: $nRes_j \leq 2^{\frac{4i-j}{2}}$
Number of communication resources	$nRes_j$	$j = 1, 3, ..., 4N - 1$; $\forall\, j, nRes_j \geq 1$ $\forall$ links left of sort: $nRes_j \leq 2^{\frac{j+1}{2}}$ $\forall$ links right of sort: $nRes_j \leq 2^{\frac{4i-j+1}{2}}$
Mapping choices	binary: m_0, m_1, m_{CR}, m_{IR}	m_0 + m_1 + m_{CR} + m_{IR} = 1; each $m_i \in top$
System-wide comm message size	2^M	M = 0, 1, ..., M_{UB} where $M_{UB} \leq N$ and $M_{UB} = K, K \in \mathbb{Z}_+$ (e.g., 14)
Sort algorithm (only with cpu_j mapping)	binary: $alg1$ or $alg2$	$m_1(fpga[0] + alg1 + alg2)$ $+(1 - m_1)(fpga[j] + alg1 + alg2) = 1, j = 2N$
Input mean job arrival rate	$\lambda_{in} \in \mathbb{R}_+$	By solving **ul**

Figure 5: Performance-related parameters of streaming sort

when every block in the application gets its own set of resources, m_1 models when all compute blocks share a single resource, m_{CR} 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 top.

Queueing network model Queuing 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*. Queuing 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 queueing 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

Figure 7: Queuing network model

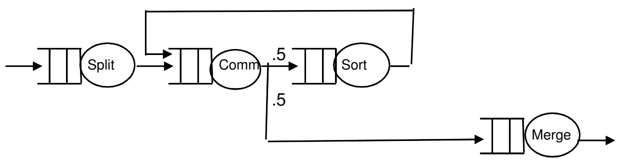

Figure 8: Queueing network has feedback when communication edges to/from sort blocks are shared

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 μ 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 validated them to be within 1% of the published empirical results [6, 7]. As an example, **l** is expressed as follows.

$$
\begin{aligned}
\lambda_j &= \lambda_{in} \quad for \quad j = 1, 2, ..., 2N-2 \\
\lambda_j &= 2\lambda_{in} \quad for \quad j = 2N-1 \\
\lambda_j &= \lambda_{in} \quad for \quad j = 2N \\
\lambda_j &= (1-m_1) \cdot \lambda_{in} \quad for \quad j = 2N+1 \\
\lambda_j &= \lambda_{in} \quad for \quad j = 2N+2, 2N+3, ..., 4N
\end{aligned} \tag{2}
$$

The expression for the mean service rate of the sort blocks is shown below. The Cs in the equation are constants.

$$
\begin{aligned}
\mu_j = \; & [cpu_0(\frac{alg_1 \times C5 \times nRes_0}{2^{B\_SZ} log(\frac{2^{B\_SZ}}{2^{\frac{j}{2}}})} \\
& + \frac{alg_2 \times C6 \times nRes_0}{2^{2B\_SZ}}) \\
& + fpga_0(\frac{C7 \times nRes_0}{2^{B\_SZ}})] \cdot [m_1] + \\
& [cpu_j(\frac{alg_1 \times C5 \times nRes_j}{2^{B\_SZ} log(\frac{2^{B\_SZ}}{2^{\frac{j}{2}}})}) \\
& + \frac{alg_2 \times C6 \times nRes_j}{2^{2B\_SZ}}) \\
& + fpga_j(\frac{C7 \times nRes_j}{2^{B\_SZ}})] \cdot [1 - m_1], \\
& j \in \{sortIndex\}
\end{aligned} \tag{3}
$$

Formulate optimization problem We use the standard *weighted sum* [15] 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 $\frac{1}{\lambda_{in}}$, the problem degenerates to identifying the *bottleneck* in the pipeline.

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

We use the equivalence property of M/M/1 BCMP networks with infinite buffers and a FIFO queueing discipline [2] to define the **mets**. Accordingly, latency is given by:

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

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.

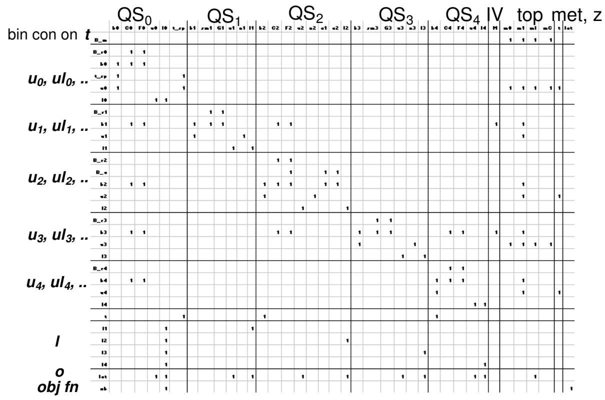

Figure 9: Matrix of variables and constraints for streaming sort

Variables other than μ, λ, $Latency$, 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

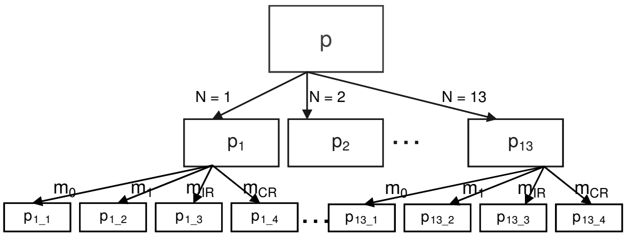

Figure 10: Decomposition of search space of streaming sort application

N	all m	m_0	m_1	m_{CR}	m_{IR}
1	305.3	305.3	1117.2	305.3	305.0
2	299.1	299.0	1737.8	426.5	299.1
3	No soln	299.0	2313.1	No soln	**298.8**
4	298.9	299.1	2849.2	No soln	298.9
5	299.1	298.9	3355.8	931.0	298.9
6	No soln	298.9	3841.4	No soln	298.9
7	298.9	No soln	4312.9	No soln	No soln
8	298.9	298.9	4772.9	No soln	No soln
9	No soln	299.1	5225.9	No soln	298.9

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 beyond $N = 3$ 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.

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 subproblems instead of just one (original) problem, each subproblem 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 nonconvex 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

Variable	Value in the solution configuration
Res. type for each compute column = FPGA	
Computation res. type determines communication res. type	
Number of resources per column ($nRes_j$ or $nRes_j$)	
$nRes_0$	1
$nRes_1$, $nRes_2$	2 each
$nRes_3$, $nRes_4$	4 each
$nRes_5$, $nRes_6$	8 each
$nRes_7$	0
$nRes_8$, $nRes_9$	4 each
$nRes_{10}$, $nRes_{11}$	2 each
$nRes_{12}$	1
M = 14 (i.e., message size = 2^{14} Bytes)	
Bottleneck mean service rate (num. of 64-bit elems per ms)	
The final merge, μ_j	10,986
Input mean job arrival rate (num. of 64-bit elems per ms)	
λ_{in}	4,513

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

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 throughout, if we increase the weight on throughput to 0.9 and reduce the weight on latency to 0.1, the recommended configuration changes from $(N = 3, m = m_{IR})$ to $(N = 4, m = m_0)$ with a corresponding objective function value of 184.5 ms.

Problem variation In 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 architecture can be shared memory or Gigabit Ethernet, but if one of the ends is mapped instead to an FPGA, the communication 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 Figure 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. The subproblems and their solutions are indexed using the same convention as above.

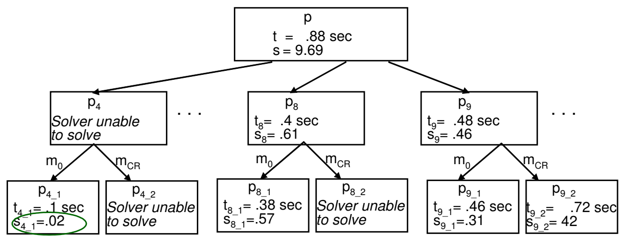

Figure 13: Results after decomposing relaxed streaming sort

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

[1] K. Abhishek, S. Leyffer, and J. T. Linderoth. FilMINT: An Outer-Approximation-Based Solver for Nonlinear Mixed Integer Programs. In *Preprint ANL/MCS-P1374-0906*, Mar. 2008.

[2] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, 1975.

[3] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, 2nd edition, 2003.

[4] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] G. Casale, M. Ningfang, and E. Smirni. Versatile models of systems using map queueing networks. In *IEEE Int'l Symp. on Parallel and Distributed Processing*, Apr. 2008.

[6] R. D. Chamberlain, G. A. Galloway, and M. A. Franklin. Sorting as a streaming application executing on chip multiprocessors. Technical Report WUCSE-2010-21, Dept. of Computer Science and Engineering, Washington University, 2010.

[7] R. D. Chamberlain and N. Ganesan. Sorting on architecturally diverse computer systems. In *Third Int'l Workshop on High-Performance Reconfigurable Computing Technology and Applications*, Nov. 2009.

[8] A. J. Coneja, E. Castillo, R. Minguez, and R. Garcia-Bertrand. *Decomposition Techniques in Mathematical Programming Engineering and Science Applications*. Springer, 2006.

[9] J. Cong, K. Gururaj, and G. Han. Synthesis of reconfigurable high-performance multicore systems. In *ACM/SIGDA Int'l Symp. on Field Programmable Gate Arrays*, pages 201–208, 2009.

[10] A. Dasgupta and R. Karri. Optimal Algorithms for Synthesis of Reliable Application-Specific Heterogeneous Multiprocessors. *IEEE Transactions on Reliability*, 44(4):603–613, Dec. 1995.

[11] M. A. Franklin, E. Tyson, J. Buckley, P. Crowley, and J. Machmeyer. Auto-pipe and the *X* language: A pipeline design tool and description language. In *Int'l Parallel and Distributed Processing Symp.*, 2006.

[12] E. Ipek, S. A. McKee, K. Singh, R. Caruana, B. R. de Supinski, and M. Schulz. Efficient architectural design space exploration via predictive modeling. *ACM Trans. Archit. Code Optim.*, 4(4):1–34, 2008.

[13] P. Krishnamurthy and R. D. Chamberlain. Analytic performance models for bounded queueing systems. In *Workshop on Advances of Parallel and Distributed Computing Models*, Apr. 2008.

[14] B. C. Lee and D. Brooks. Roughness of microarchitectural design topologies and its implications for optimization. In *IEEE High Performance Computer Architecture*, pages 240–251, Feb 2008.

[15] R. Marler and J. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, 2004.

[16] NEOS Server and Solvers for Optimization. http://neos.mcs.anl.gov/neos/solvers/index.html.

[17] A. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. on Computers*, 55(2):99–112, Feb. 2006.

[18] G. Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 1980.

[19] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. *11th Int'l Conf. on Compiler Construction*, pages 179–196, 2002.

[20] B. W. Wah and Y. Chen. Solving large-scale nonlinear programming problems by constraint partitioning. In *11th Int'l Conf. on Principles and Practice of Constraint Programming*, pages 697–711, Oct. 2005.

[21] H. Youn, S. Jang, and E. Lee. Deriving Queuing Network Model for UML for Software Performance Prediction. In *5th ACIS Int'l Conf. on Software Engineering Research, Management & Applications*, pages 125–131, 2007.