Report Number: WUCSE-2010-58

2010

# Optimal Design-space Exploration of Streaming Applications

Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain

Many embedded and scientific applications are frequently pipelined asynchronously and deployed on architecturally diverse systems to meet performance requirements and resource constraints. We call such pipelined applications streaming applications. Typically, there are several design parameters in the algorithms and architectures used that, when customized, impact the tradeoff between different metrics of application performance as well as resource utilization. Automatic exploration of this design space is the goal of this research. When using architecturally diverse systems to accelerate streaming applications, the design search space is often complex. We present a global optimization framework comprising a novel domain-specific variation of branch-and-bound... **Read complete abstract on page 2.**

# Optimal Design-space Exploration of Streaming Applications

Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain

Complete Abstract:

Many embedded and scientific applications are frequently pipelined asynchronously and deployed on architecturally diverse systems to meet performance requirements and resource constraints. We call such pipelined applications streaming applications. Typically, there are several design parameters in the algorithms and architectures used that, when customized, impact the tradeoff between different metrics of application performance as well as resource utilization. Automatic exploration of this design space is the goal of this research. When using architecturally diverse systems to accelerate streaming applications, the design search space is often complex. We present a global optimization framework comprising a novel domain-specific variation of branch-and-bound that reduces search complexity by exploiting the topology of the application's pipelining. We exploit the topological information to discover decomposability through the canonical Jordan block form and to originate two novel decomposition techniques that we name linear chaining and convex decomposition. The reduction in search complexity that we achieve for two prototypical real-world streaming applications is significant. For one application the size of the search space reduces from about 10^18 to 10^12, a million-fold reduction, and for the second application, the reduction is a factor of at least 20~million.

2010-58

# Optimal Design-space Exploration of Streaming Applications

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

Corresponding Author: spadmanabhan@wustl.edu

Abstract: Many embedded and scientific applications are frequently pipelined asynchronously and deployed on architecturally diverse systems to meet performance requirements and resource constraints. We call such pipelined applications streaming applications. Typically, there are several design parameters in the algorithms and architectures used that, when customized, impact the tradeoff between different metrics of application performance as well as resource utilization. Automatic exploration of this design space is the goal of this research.

When using architecturally diverse systems to accelerate streaming applications, the design search space is often complex. We present a global optimization framework comprising a novel domain-specific variation of branch-and-bound that reduces search complexity by exploiting the topology of the application's pipelining. We exploit the topological information to discover decomposability through the canonical Jordan block form and to originate two novel decomposition techniques that we name linear chaining and convex decomposition. The reduction in search complexity that we achieve for two prototypical real-world streaming applications is

Type of Report: Other

# Optimal Design-space Exploration of Streaming Applications

Shobana Padmanabhan, Yixin Chen, and Roger D. Chamberlain
Dept. of Computer Science and Engineering, Washington University in St. Louis
{spadmanabhan, ychen25, roger}@wustl.edu

***Abstract*—Many embedded and scientific applications are frequently pipelined asynchronously and deployed on architecturally diverse systems to meet performance requirements and resource constraints. We call such pipelined applications streaming applications. Typically, there are several design parameters in the algorithms and architectures used that, when customized, impact the tradeoff between different metrics of application performance as well as resource utilization. Automatic exploration of this design space is the goal of this research.**

**When using architecturally diverse systems to accelerate streaming applications, the design search space is often complex. We present a global optimization framework comprising a novel domain-specific variation of branch-and-bound that reduces search complexity by exploiting the topology of the application's pipelining. We exploit the topological information to discover decomposability through the canonical Jordan block form and to originate two novel decomposition techniques that we name linear chaining and convex decomposition. The reduction in search complexity that we achieve for two prototypical real-world streaming applications is significant. For one application the size of the search space reduces from about $10^{18}$ to $10^{12}$, a million-fold reduction, and for the second application, the reduction is a factor of at least 20 million.**

## I. Introduction

High performance streaming applications are pipelined asynchronously and frequently deployed on architecturally diverse systems (employing chip multiprocessors, graphics engines, and reconfigurable logic) [8], [13]. Typically, there are several design parameters (examples in Section II) in the algorithms and architectures used that, when customized, impact the tradeoff between application performance and resource utilization. Searching the design space of possible configurations resulting from varying the parameters is hard because: (1) the number of configurations is exponential in the number of design parameters, (2) the design parameters may interact nonlinearly, and (3) goals of the design-space exploration are often multiple and conflicting. Though this problem has been researched for system-on-chip applications, to our knowledge, we are the first to focus on it for streaming applications.

For embedded applications, design-space exploration has been researched using search heuristics as well as standard and modified optimization techniques [12], [15], [29]. 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 [21], [23] and examples of search heuristics include gradient ascent [23]. 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.

We approach design space exploration as an optimization problem. We exploit queueing network (QN) models [4] because they embody the queueing and topology of our application's pipelining (e.g., see [7]) better than general-purpose models based on regression or machine-learning [21], [23]. We derive our objective function using the standard technique of weighted sum of normalized cost functions [25]; constraints may stem from the design space, QN models, resource availability, performance requirements, and the optimization problem formulation itself. The solution from solving the optimization problem is a recommended configuration (i.e.) recommended value for each design parameter.

Such optimization problems tend to be NP-hard because: (1) most design parameters are integer-valued, (2) QN expressions are nonlinear, and (3) the nonlinear functions (objective function or constraints) are typically not convex, differentiable, or even continuous. Worse, in practice, state-of-the-art solvers such as Bonmin or FilMINT [26] often fail to find even a feasible solution. Hence, we have developed a search framework, based on the branch and bound principle, that systematically finds the global optimum given adequate time.

With standard branch and bound, efficient bounding helps to prune branches which makes the search more efficient. However, the optimization problems in our domain of streaming applications tend to be highly nonlinear and mostly discrete and hence, none of the standard relaxation techniques [5], [11] work, as we verified for our two applications. This implies we cannot bound and hence cannot prune branches. Hence, the efficiency of our search depends on the ordering of branching variables (BVs) and identifying properties and conditions that obviate having to evaluate every branch. To help us here, we exploit the topology of the application pipelining.

Consider the example of Figure 1. A three-stage pipelined application has been modeled with a three-stage queueing network. When the buffering capacity within the application is sufficient to handle the queueing requirements, queueing theory tells us that we can reason about the three individual queues separately, and the only required interaction is via the flow rates of jobs (modeling data elements) between the

Sponsored by NSF under grants CNS-0905368 and CNS-0931693.

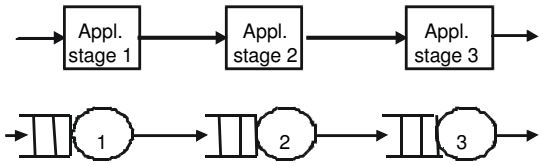


Fig. 1: Example streaming application and associated queueing network.

queueing stations.

In the context of an optimization problem, the topology of the queueing network can give us significant clues as to how to efficiently search the design space. In this paper, we propose a number of techniques to do precisely that.

Our main contributions are: (1) We identify and categorize domain-specific topological information. (2) Using the categories, we develop a heuristic that orders the BVs such that each branching leads to maximum decomposition of the search space. We introduce a way to identify decomposability of our applications based on the canonical Jordan block form (JBF). (3) To improve our search efficiency further, we originate two decomposition techniques that we name *linear chaining* and *convex decomposition*. (4) Our framework supports anytime solutions if application developers need a suboptimal solution fast. (5) We exemplify the application and benefit of using our decomposition techniques using two prototypical real-world streaming applications.

The rest of the paper is organized as follows. We introduce the streaming applications that we use and their queueing network performance models and cost functions in Section II. We present the details of our main contributions in Section III. In Section IV, we provide empirical results that demonstrate the use of our decomposition techniques and show a benefit of at least a million-fold reduction in the size of the search space.

## II. Background

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 $M$ indicates 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 [4]. In such a network, it is conventional to define $\lambda_j$ as the mean arrival rate and $\mu_j$ as the mean service rate for queueing station $j$. Here, we restrict ourselves to $M/M/1$ BCMP networks with infinite buffers and a FIFO queueing discipline, or (when not BCMP) use the techniques described in [22], [28] to model bounded buffer networks with upstream blocking. 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., [22] for approaches to handling a wider class of service distributions.

Analytic models for the service rates and network branching probabilities are assumed to come from the user or developed using general-purpose empirically-driven techniques [21], [23].

We illustrate our approach using examples from two well-known classes of applications—data-filtering applications and data-decomposing applications. In the former class, application data gets filtered as it travels down the pipeline. In the latter, incoming data is divided, processed in parallel for accelerated performance, and the results combined. Biosequence similarity search [6] is the example we use from the former class and stream-based sorting [9], [10] is the example from the latter. Each of these two applications, their respective queueing network-based performance models, and their cost functions for optimization are described next.

### A. BLAST

BLAST, the Basic Local Alignment Search Tool, is the leading algorithm for searching genomic and proteomic sequence data [2], [3]. BLASTN, the variant that focuses on genomic data, has been accelerated both on multiprocessor architectures [14], [24] and using special-purpose hardware [6], [16], [20]. BLASTN is an example of a data-filtering application, a series of pipelined computations that each progressively filter a fraction of the input data, resulting in a heuristic search that is both fast and effective.

The BLASTN search heuristic compares a query string (composed from the alphabet $\{A, T, G, C\}$) to a genomic database, looking for biologically significant approximate matches. The pipeline structure of the accelerated implementation that we experimented with is illustrated in Figure 2. After the loading of a query string, the database is streamed in from the left of the figure. The database moves across the PCI-X bus into an FPGA and enters stage 1a (which is decomposed internally into a varying number of substages, 1a1 to 1a5 or 1a6 depending upon the parallelism built into the Bloom filter). Database entries that match in the Bloom filter move to stage 1b, where they are checked against a hash table built from the query string. If found in the hash table, hits move downstream to stage 2, where they are filtered and hits that survive stage 2 filtering are moved back across the PCI-X bus to stage 3.

A queueing network-based performance model of this BLASTN implementation was developed and validated by Dor et al. [17], and the model used here is similar in structure. The queuing network is illustrated in Figure 3. Station 0 models the PCI-X bus, which handles data both to and from the FPGA with a single server because the PCI-X bus is a half-duplex bus. Stage 1a is modeled using the queueing stations 1a1 through 1a5 or 1a6. Station 1a represents a bank of Bloom filters, and stations 1a2 and up model the combining network that aggregates the hits out of the Bloom filter.

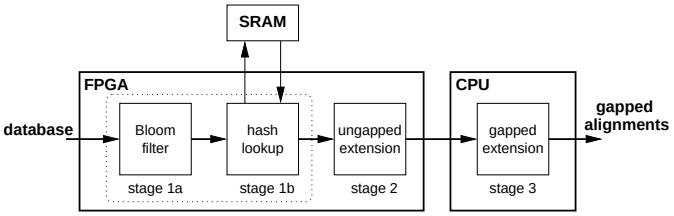


Fig. 2: Pipeline structure of BLASTN application [6].

| Variable | Symbol | Ranges and Constraints |
|---|---|---|
| Clock freq. (stages 1a, 1b, 2) | $f_{1a}$,$f_{1b}$,$f_2$ | $10 \leq f \leq 133.3$ MHz |
| Processor cores (stage 3) | $c$ | $1 \leq c \leq 4$ |
| Bloom filter hash functions | $k$ | $2 \leq k \leq 10$ |
| Bloom filter memory size | $m$ | $1000 \leq m \leq 2000$ |
| Query length | $q$ | $40,000 \leq q \leq 65,000$ |
| Word size | $w$ | $10 \leq w \leq 13$ |
| Buffer size | $b_i$ | $2 \leq b_i \leq 16$, $1 \leq i \leq r$ |
| Reduction tree size | $r$ | 5 or 6 |
| Stage 2 threshold | $p_2$ | $10^{-8} \leq p_2 \leq .005$ |
| Input mean job arrival rate | $\lambda_{in} \in \mathbb{R}_+$ | By solving **ul** |

Fig. 4: Design variables for BLASTN.

Whether or not there is a stage 1a6 depends upon the number of Bloom filters instantiated in the design. This is one of the design parameters to be explored during the optimization process. The remainder of the stages and queueing stations are easily aligned.

The set of design variables (corresponding to the design parameters), along with their characterization, bounds and constraints, for BLASTN are enumerated in Figure 4. We wish to optimize the combination of throughput (measured by $\lambda_{in}$) and FPGA power consumption. With $p_i$ denoting the probability that a stage passes an input to its downstream neighbor, the relationship between the arrival rates at each stage are

$$\begin{aligned} \lambda_{1a1} &= \lambda_{in} \\ \lambda_{1a2} &= 2 \cdot p_{1a} \cdot \lambda_{1a1} \\ \lambda_{1a,j} &= 2 \cdot \lambda_{1a,j-1} \quad for \quad 3 \leq j \leq r \\ \lambda_{1b} &= \lambda_{1a,r} \\ \lambda_2 &= p_{1b} \cdot \lambda_{1b} \\ \lambda_3 &= p_2 \cdot \lambda_2 \\ \lambda_0 &= \lambda_{in} + \lambda_3 \end{aligned} \tag{1}$$

Within stage 1a, which requires a bounded buffer size model, the mean service rate of an upstream node is scaled by the queue occupancy of the downstream node, e.g.,

$$\mu_{1a1} = \left(1 - \left(\frac{\lambda_{1a2}}{\mu_{1a2}}\right)^{b_2}\right) \cdot f_{1a} \tag{2}$$

The power is modeled in the traditional manner as a combination of dynamic power (linearly related to clock frequency) and static power (a constant, independent of clock frequency). This gives a power equation of the form

$$P = m_{1a} \cdot f_{1a} + m_{1b} \cdot f_{1b} + m_2 \cdot f_2 + P_{static} \tag{3}$$

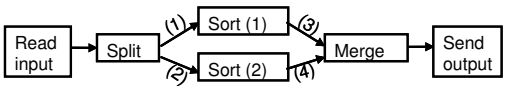


Fig. 5: A streaming sort application.

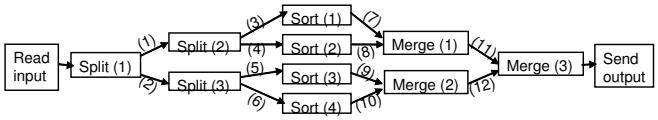


Fig. 7: A streaming sort application with 4 sort blocks.

where the values for $m_{1a}$, $m_{1b}$, $m_2$, and $P_{static}$ were derived from Xilinx Power Estimator, ISE v11.5i.

The resulting cost function is a weighted sum of the individual optimization goals, i.e.,

$$maximize \quad W_1 \times \lambda_{in} - W_2 \times P \tag{4}$$

where $W_1$ and $W_2$ encode both the weights and normalization factors.

As described above, the optimization problem is a mixed-integer nonlinear problem (MINLP). It has 12 integer-valued variables and 4 continuous variables, and state-of-the-art solvers (FilMINT) fail to find a feasible solution.

### *B. Streaming Sort*

Our second example application is a streaming implementation of sorting. While sorting is used extensively, our specific use case is drawn from the field of computational finance. In computing the value-at-risk for a portfolio of financial instruments, there is a repeated requirement to sort one million elements at high throughput with minimum latency [8], [30]. The effective use of architecturally diverse platforms for sorting has received considerable attention in the literature [10], [18], [19].

In our streaming sort, input data is *split* into parts and each part is sent to a *sort* instantiation. This application topology is shown in Figure 5. The sort instantiation is referred to as a sort "block" in the parlance of Auto-Pipe [8]. 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 5.

The set of design variables, along with their characterization, bounds and constraints, for streaming sort are enumerated in Figure 6. 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 changes to the application topology as illustrated in Figure 7.

For the sorting application, the queueing model was introduced earlier [27]. It begins with the application topology shown in Figure 5. The queueing network model for this

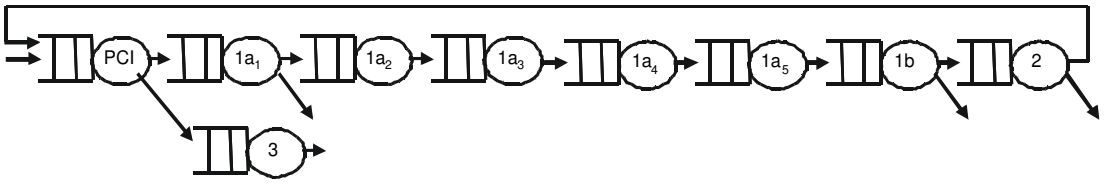


Fig. 3: Queueing network-based performance model of BLASTN application.

| 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, ..., $\frac{B\_SZ}{2}$; $N \in top$ (see section III) |
| Index of split columns<br>of link columns left of sort column<br>of sort column<br>of link columns right of sort column<br>of merge columns | $j$ | $j = 0, 2, ..., 2N-1$<br>$j = 1, 3, ..., 2N-1$<br>$j = 2N$<br>$j = 2N+1, 2N+3, ..., 4N-1$<br>$j = 2N+2, 2N+4, ..., 4N$ |
| 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 | $nRes_j$ | $j = 0, 2, ..., 4N$; $\forall j$, $nRes_j \geq 1$<br>$\forall$ split columns: $nRes_j \leq 2^{\frac{j}{2}}$; $\forall$ sort columns: $nRes_j \leq 2^{\frac{j}{2}}$<br>$\forall$ merge columns: $nRes_j \leq 2^{\frac{4i-j}{2}}$ |
| Number of communication resources | $nRes_j$ | $j = 1, 3, ..., 4N-1$; $\forall\ j$, $nRes_j \geq 1$<br>$\forall$ links left of sort: $nRes_j \leq 2^{\frac{j+1}{2}}$<br>$\forall$ 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 | $2^M$ | $M = 0, 1, \ldots, M_{UB}$ where $M_{UB} \leq N$ and $M_{UB} = K, K \in \mathbb{Z}_+$ (e.g., 14) |
| Sort algorithm (only with $cpu_j$ mapping) | binary: $alg_1$ or $alg_2$ | $m_1(fpga_0 + alg_1 + alg_2)$<br>$+(1-m_1)(fpga_j + alg_1 + alg_2) = 1, j = 2N$ |
| Input mean job arrival rate | $\lambda_{in} \in \mathbb{R}_+$ | By solving **ul** |

Fig. 6: Design variables for streaming sort. Binary variables are used to select among resource types (CPU vs. FPGA), mappings, and sort algorithms.


Fig. 8: Queuing network model.

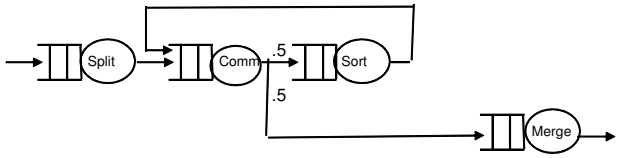


Fig. 9: Queueing network has feedback when communication edges to/from sort blocks are shared.

topology is shown in Figure 8. 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 9, where the server "Comm" is handling all of the communication both into and out of the "Sort" server.

The expressions for mean service rate at each server, $\mu_j$, and the relationships between the mean arrival rates, $\lambda_j$, are derived from first principles and have been validated to be within 1% of published empirical results [9], [10]. For example, arrival rates at each queueing station are related as follows.

$$\begin{aligned} \lambda_j &= \lambda_{in} \quad for \quad j = 1, 2, ..., 2N-2 \\ \lambda_j &= 2\lambda_{in} \quad for \quad j = 2N-1 \\ \lambda_j &= \lambda_{in} \quad for \quad j = 2N \\ \lambda_j &= (1-m_1)\cdot\lambda_{in} \quad for \quad j = 2N+1 \\ \lambda_j &= \lambda_{in} \quad for \quad j = 2N+2, 2N+3, ..., 4N \end{aligned} \tag{5}$$

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

$$\begin{aligned} \mu_j = [&cpu_0(\frac{alg_1 \times C_5 \times nRes_0}{2^{B\_SZ}\log(\frac{2^{B\_SZ}}{2^{\frac{j}{2}}})} \\ &+\frac{alg_2 \times C_6 \times nRes_0}{2^{2B\_SZ}}) \\ &+fpga_0(\frac{C_7 \times nRes_0}{2^{B\_SZ}})] \cdot [m_1] + \\ &[cpu_j(\frac{alg_1 \times C_5 \times nRes_j}{2^{B\_SZ}\log(\frac{2^{B\_SZ}}{2^{\frac{j}{2}}})} \\ &+\frac{alg_2 \times C_6 \times nRes_j}{2^{2B\_SZ}}) \\ &+fpga_j(\frac{C_7 \times nRes_j}{2^{B\_SZ}})] \cdot [1-m_1], \\ &j \in \{sortIndex\} \end{aligned} \tag{6}$$

As motivated by [30], the performance objectives for our streaming sort application include both minimizing latency and maximizing throughput. We use the standard weighted sum [25] technique to combine the multiple (normalized) performance objectives as shown in equation 7. 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.

$$minimize \quad W_1 \times Latency + W_2 \times \frac{1}{\lambda_{in}}, \sum_{1}^{2} W_i = 1 \qquad (7)$$

From queueing theory (for $M/M/1$ BCMP networks), latency is given by:

$$Latency = \sum_{j=0}^{4N} \frac{1}{\mu_j - \lambda_j} \qquad (8)$$

As the above illustrates, 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. Variables other than $\mu$, $\lambda$, and $Latency$ are integer-valued, making the problem mixed-integer.

In what follows, we assume that queueing network performance models similar in style to the two above examples are provided, and the task is to search the design space of the application for an optimal (or anytime suboptimal) configuration. Our techniques exploit the topological information present in the queueing network models to improve the efficiency of the design space exploration.

## III. Domain-specific optimization

Branching on a variable decomposes the search space resulting in subproblems that are less complex. Our aim then is to order the BVs such that the current branching leads to more decomposition of the search space than the subsequent one. We identify decomposability from variables by checking if a matrix showing the presence of variables in the constraints and the objective function has the canonical JBF [31]. In this form, the objective function and the constraints separate into blocks that can be solved independent of each other without loosing optimality.

In real-world applications, there are usually variables that prevent such decomposition. Such variables are traditionally called complicating variables (CVs) (Figure 10). Decomposition techniques exist for dealing with CVs [11], [5]. 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 complicating constraints (CCs) and apply the outer linearization algorithm, provided

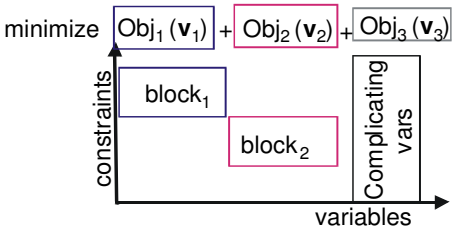


Fig. 10: Jordan block form with complicating variables.

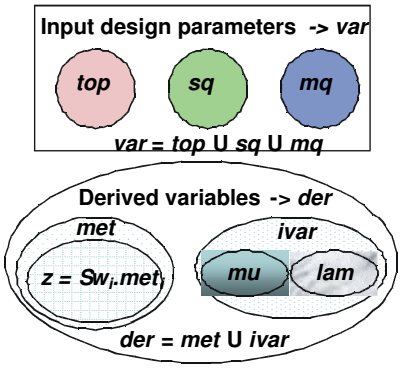


Fig. 11: Categories of design variables.

the nonlinear constraints are inequalities and the objective function is linear (both of which can often be achieved through simple transformations). Here, we describe a domain-specific decomposition technique that exploits the topological structure of the queueing network performance model.

For our applications, we identify JBF using the BCMP independence property which states that the service rates of the queueing stations (QSs) in the QN are independent of each other. In other words, the QSs in a BCMP QN are related only through job arrival rates. In this case, we group variables and constraints associated with only a single QS as a Jordan block (JB). Hence, variables concerning more than a single QS would be CVs. We define the *degree of complication* as the number of QSs a CV concerns so that the *most complicating variable* has the highest $degree$. First, we identify and categorize the application's topological information such that the categories help us identify the CVs and their degrees of complication. These categories were introduced earlier [27] and are summarized in Figure 11.

### A. Domain-specific topological info

In the categorization below, variables in $top$ are the most complicating, followed by those in $mj$, whereas $sj$ variables are not complicating. If $\lambda_{in} \in var$, by **l**, $\lambda_{in}$ has the highest degree of complication in $mj$.

1) $var = \{var_i | var_i \in \mathbb{R}_+ \cup \{0\}\}$ is the set of design variables in the optimization problem that correspond to the user-identified design parameters. $n_v \doteq |var|$ and $\mathbf{v} \in (\mathbb{R}_+ \cup \{0\})^{n_v}$ is the vector of the variables in $var$. $var = top \cup mj \cup sj$.

a) $top \subseteq var$ is the set of topological variables that result in distinct alternative QN topologies. $n_t \doteq |top|$ and **t** is the vector formed by the variables. We discovered the presence of $top$ during our experimentation (i.e.) they are not traditional.
b) $mj \subseteq var$ is the set of Multi-JB variables, with each element in the domain of more than one $u_j(\cdot)$. $n_{mj} \doteq |mj|$.
c) $sj \subseteq var$ is the set of Single-JB vars. Each element of $sj$ is in the domain of only one $u_j(\cdot)$. $n_{sj} \doteq |sj|$.

2) $der = \{der_i | der_i \in \mathbb{R}_+ \cup \{0\}\}$ is the set of derived variables that depend on one or more elements in $var$. $der \bigcap var = \emptyset$.

a) $met \subseteq der$ is the set of performance metrics. $n_m \doteq |met|$ and **met** is the vector formed by the variables in *met*. $met_k = o_k(\mathbf{v}) : (\mathbb{R}_+ \cup \{0\})^{n_v} \rightarrow \mathbb{R}_+ \cup \{0\}$ where **o** is the vector of functions that define **met**.
b) $z \in der$ is the cost function (the objective function). $z = \sum_{k=1}^{n_m} W_k \times met_k : (\mathbb{R}_+ \cup \{0\})^{n_m} \rightarrow \mathbb{R}_+, \sum_{k=1}^{n_m} W_k = 1$, $W_k$ are the weights.
c) $ivar \subseteq der$ is the set of intermediary variables (IVs). IVs may arise because: (1) application developers are interested in them (e.g., for debugging) (2) to codify abstractions such as QNs in the performance models and (3) to help optimization solvers (cutting-plane based solvers such as FilMINT [1] tend to work better with linear cost functions while solvers using the interior-point algorithm work better with linear constraints [5]).

i) $mu \subseteq ivar$ is the set of mean service rates at each QS. $\mu$ is the vector formed by the variables in *mu*. $\mu_j = u_j(\mathbf{v}) : (\mathbb{Z}_+ \cup \{0\})^{n_v} \rightarrow \mathbb{R}_+$ where **u** is the vector of functions that define $\mu$.
ii) $lam \subseteq ivar$ is the set of mean job arrival rates at each QS. $\lambda$ is the vector formed by the variables in *lam*. $\lambda$ are related by a system of linear equations with a unique solution (in terms of input mean job arrival rate denoted by $\lambda_{in} \in \mathbb{R}_+$). $\lambda_j = l_j(\lambda_{in}, \mathbf{t}) : \mathbb{R}_+ \cup \{0\} \times \mathbb{Z}_+^{n_t} \rightarrow \mathbb{R}_+ \cup \{0\}$ where **l** is the vector of functions that define $\lambda$.
iii) **ul** is a vector of QN constraints that restricts every $\lambda_j < \mu_j$ for the system to be stable.

Queueing network topologies are annotated digraphs. Annotations include, at the minimum, expressions for each of **u**, **l**, **o**, and $z$; each node represents a QS which is a service facility with its queue and each edge, the communication link between the two connected nodes. We identify the general form of our optimization problem to be the following. **g** and **h** are vectors of the general constraints.

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

Specific problems that are not originally provided in the above form (e.g., the BLASTN application) are readily transformed into this form using standard techniques.

### *B. Heuristic to order branching variables*

We order the BVs in the order of decreasing degree of complication, first across variable categories, and then within a category. We break ties within a category in favor of the variable with the largest domain. In addition to the tie-breaking rules presented earlier [27], if there is more than one variable with the same number of possible values, we favor the variable that reduces the complexity of the most number of functions (the objective function and the constraints). The resulting ordering is:

1) Branch on $top$ variables. After branching on all $top$ variables, each branch will evaluate only one topology, by definition of $top$.
2) Branch on $\lambda_{in}$ if it is in $var$.
3) Branch on the remaining $mj$ variables. After branching on all $mj$, each subproblem concerns only one queueing station because $var - top - mj = sj$.
4) If the subproblems are still unsolvable (e.g., $u$ is not convex), branch on $sj$ variables. The solution after branching on all of $sj$ is the global optimum since $var = top + mj + sj$.

**Reduction in number of branch evaluations:** If there is more than one $sj$ per JB, then, their combinations inside a JB need to be evaluated. However, by our definition of $sj$ variables, $sj$ variables across JBs are independent of each other and hence their combinations need not be evaluated (i.e.) the subproblems corresponding to the combinations need not be solved. This leads to a significant reduction in the number of branches that need to be evaluated. For example, consider $n_v$ variables with each variable having $k$ values, the total number of nodes in the branch and bound tree, including the root, is $\frac{1-k^{n_v+1}}{1-k}$. On the other hand, by our definition of $sj$ variables, the complete enumeration need only happen for $n_t + n_{mj}$ variables (recall that $n_v = n_t + n_{mj} + n_{sj}$). To count the number of branch evaluations involving $sj$ variables, let the number of distinct Jordan blocks be denoted by $J$ and let the variables in $sj$ be evenly divided among the Jordan blocks. Then, the number of variables in each Jordan block is given by $n_{sj}/J$, and the total number of nodes in the reduced branch and bound tree is

$$\frac{1-k^{n_t+n_{mj}+1}}{1-k} + k^{n_t+n_{mj}} \left( \frac{1-k^{n_{sj}/J+1}}{1-k} - 1 \right).$$

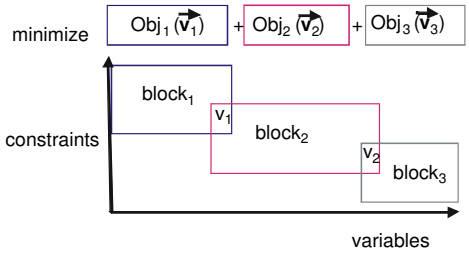


Fig. 12: Linear chaining of Jordan blocks.

This represents a significant savings anytime that $J >> 1$.

### C. Linear chaining

Based on our experiments, we discovered that when there is one and only one complicating variable that connects two neighboring Jordan blocks, we call those JBs *linearly chained*. We call such a CV a *chain-CV*. This linear chaining occurs when the corresponding queueing stations violate the BCMP independence property (i.e.) their service rates are not independent. This is illustrated in Figure 12, where variable $v_2$ is dependent upon block 3 and also influences block 2. In a similar manner, variable $v_1$ is dependent upon block 2 and influences block 1. In the presence of this structure and if the objective function value has been determined (by virtue of branching on all the variables that form the objective function), we can recast the problem of the right-most JB (in the ordering of Figure 12) into two problems—one to maximize the chain-CV and the other to minimize the chain-CV. The solutions from the two subproblems form the upper and lower bound of the chain-CV in the problem corresponding to the immediate left JB. The same procedure is repeated until we reach the left-most JB in the chain at which point we would obtain the feasible solution that corresponds to the objective function value.

**Reduction in number of branch evaluations**: By exploiting the linear chain structure, we avoid evaluating the exhaustive combinations of all the variables concerning the JBs in the chain. This leads to another significant reduction in the number of branches evaluated as illustrated for the BLASTN application in Section IV-A.

### D. Convex decomposition

Based on our experiments, we also discovered a convexity property which when present obviates evaluating every value of a branching variable: *if the branching variable (BV)* $\in \mathbb{R}$ *and if the objective function of the BV is convex, we can solve for that variable analytically* (by setting the first derivative of $z$ w.r.t. the BV $= 0$). We name this decomposition *convex decomposition* and the variable exhibiting this property *convex-BV*.

An example of the form of objective function that will enable such convenient decomposition is presented for stream-based sorting application in Section IV-B [27]. We clarify our preliminary definition of convex decomposition [27], by stating that $met$ need to be functions of a convex-BV such that when branching on that variable, the problem decomposes to not include the variable.

**Reduction in number of branch evaluations**: Convex decomposition can lead to a large reduction in the number of branches that need to be evaluated because such variables are continuous and hence, depending on the granularity of discretization, we could potentially have thousands of possible values for a single variable. This property is very useful as there are many applications that are modeled using BCMP QNs with similar cost functions.

### E. Search procedure

Our search procedure is: (1) choose and branch on the next BV; (2) solve the resulting subproblems (exploiting the chaining and convexity properties if present) and update the incumbent solution (initialized to $\infty$ for a minimization problem); and (3) terminate if applicable or go back to (1). We compare the incumbent solution against the objective function value in every subproblem resulting from the current branching. If a subproblem is *convex* and its solution is worse than the incumbent solution, we stop branching on that subproblem. If every subproblem from a branching is convex, we stop branching, and the incumbent solution, updated as applicable, is the global optimum.

In many circumstances, it is not necessary to determine a globally optimal configuration. It may be sufficient to find a feasible solution, or to find a feasible solution with a cost function that is below some input threshold. In such circumstances, it is possible to terminate the search prior to the determination of a global optimum. The search procedure above supports these anytime solutions, in which early termination can still yield one (or more) feasible configurations.

## IV. Empirical results

As we branched on design variables and solved the resulting subproblems, we determined that many of the solutions, when found and reported by the solver, were often not optimal even locally. This is because the optimization problem formulations of our streaming applications are highly nonlinear and mostly discrete and hence remain nonconvex through many levels of branching. This is also the reason why the standard relaxation techniques did not work for our optimization problems, making it not possible to bound and prune branches. The state-of-the-art solver that we used for solving our optimization problem formulations is FilMINT.

### A. BLAST

*1) Variable categorization:* We begin by categorizing the design variables described in Section II-A, according to the categories we introduced in Section III-A. $r = 6$ in the

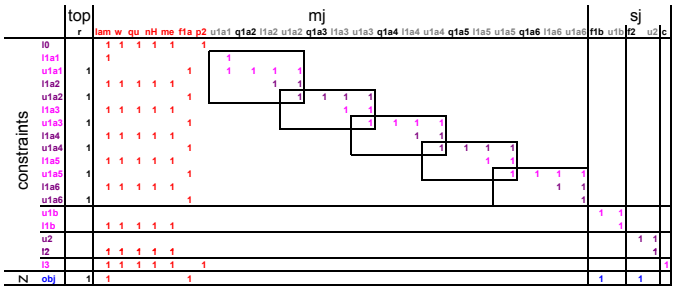


Fig. 13: BLASTN variable-constraint matrix.

following equations.

$$\begin{aligned} top &= \{r\} \\ mj &= \{\lambda_{in}, k, q, w, m, p_2\} \\ sj &= \{b_{1a_2}, b_{1a_3}, \ldots b_{1a_6}, f_{1b}, f_2, c\} \\ mu &= \{\mu_0, \mu_{1a_1}, \ldots \mu_{1a_6}, \mu_{1b}, \mu_2, \mu_3\} \\ lam &= \{\lambda_0, \lambda_{1a_1}, \ldots \lambda_{1a_6}, \lambda_{1b}, \lambda_2, \lambda_3\} \\ met &= \{throughput, power\} \end{aligned}$$

*2) Variable-constraint matrix:* To illustrate the structure that is present in the optimization problem that is exposed by the queueing topology, Figure 13 shows the variable-constraint matrix for the BLASTN application. Each column corresponds to an individual variable (both design variables and derived variables) in the problem formulation, and each row corresponds to an individual constraint. The matrix is a Boolean-valued matrix, with a 1 in an entry if the variable associated with the column is present in the constraint associated with the row. The columns are ordered by the search procedure described in Section III-E.

This ordering of columns puts the complicating variables in the leftmost columns. The first thing to note is how few of the columns (i.e., variables) are complicating across the majority of the constraints. Second, note that the majority of the middle columns comprise a set of linearly chained Jordan blocks. The set of equations that relate the $\mu$'s associated with application stage 1a give rise to this linear chain, and these $\mu$'s are chain-CVs. An example of one of these equations is (2) in Section II-A. Finally, on the far right are three blocks of Single-JB variables associated with stages 1b, 2, and 3, respectively. Overall, the intuitive notion one gets from observing this matrix is that if the structure exposed in the matrix can be exploited during the design space search process, clear benefits will accrue.

*3) Branch and bound:* For our optimization problem formulation for BLASTN (based on the provided performance models discussed in Section II-A), the state-of-the-art solver that we use did not find even a feasible solution. In our branch and bound search, we ordered the branching variables according to the heuristics articulated in Section III.

We started with branching on $r \in top$ and the solver did not find a feasible solution for any of the resulting subproblems. Second, we branched on $\lambda_{in} \in mj$. $\lambda_{in}$ is not a convex branching variable for this problem because $\lambda_{in}$ does not decompose the problem. This is because while $throughput \in met$ benefits from increased service rates of the queueing stations, $power \in met$ does not. This lead to having to branch on $\lambda_{in} \in \mathbb{R}$. One way to branch on continuous variables is to "discretize" the range. For $\lambda_{in}$, we calculated its upper bound based on possible services rates. The resulting subproblems at this level are still not handled by the solver, mainly because the functions relating Bloom filter-related variables for $p_i$ still remain non-convex. Thirdly, we branched on $k \in mj$. Here, the solver did find a feasible solution for the subproblem corresponding to $k = 3$. We then branched successively on $q \in mj$, $w \in mj$, $m \in mj$, and $p_2 \in mj$. The subproblems remained largely unsolved through these branchings.

The only remaining $mj$ variable at this level of branching is $f_{1a} \in \mathbb{R}$ and it chains stages $1a_1$ through $1a_6$ (through the $\mu$'s described above). Exploiting this linear chain ends up decomposing the problem such that the $sj$ variables concerning the above stages form JBF and hence can be solved independently from each other. The involved $sj$ variables are $b_{1a_2}, b_{1a_3}, \ldots b_{1a_6}$. When we consider $15$ discrete values for each of these variables, the number of subproblems we need to evaluate is $15 \times 5 \times 2 = 150$ rather than the otherwise required $15^5 = 759,375$ subproblems. This represents more than three orders of magnitude reduction in the number of branches evaluated just at this level.

All the remaining variables at this level are in $sj$ and they are $f_{1b} \in \mathbb{R}$, $f_2 \in \mathbb{R}$, and $c$. Even if we evaluate only 100 discrete values for each of $f_{1b}$ and $f_2$, the number of branches evaluated will be $204$ rather than the otherwise required $40,000$. In reality, we might consider thousands of discrete values for each continuous variable which implies even greater savings in the number of evaluations.

In total, a completely enumerated branch and bound search of the design space requires $72,000,000 \times 759,375 \times 40,000 \approx 2 \times 10^{18}$ evaluations (assuming only 100 evaluations for each continuous variable, a very conservative assumption). Our technique reduces this to $72,000,000 \times 150 \times 204 \approx 2 \times 10^{12}$ evaluations, a reduction of a million fold.

The objective function value of the global optimum is $62.6$ which corresponds to the configuration of Figure 14. The properties of this configuration are consistent with what we know about the system that has been physically constructed [6]. Stage 1a is the bottleneck stage, and the configuration of Figure 14 gives it a high clock frequency. The latter stages have much less work to perform, making their lower clock frequency beneficial. Note that the implementation in [6] made no attempt to simulaneously lower power consumption, so we would not expect the to configurations to be identical.

### B. Streaming sort

*1) Variable categorization:* A subset of the details of the categorization of the design variables described in Section II-B, according to the categories we introduced in Section III-A was presented earlier [27]. Here $N = 1$ (i.e.) two

| Variable | Value in the solution configuration |
|---|---|
| $r$ | 5 |
| $\lambda_{in}$ | 132 MBases/sec |
| $k$ | 3 |
| $q$ | 40,000 bases |
| $w$ | 10 bases |
| $m$ | 1000 |
| $p_2$ | $4.9 \times 10^{-8}$ |
| $f_{1a}$ | 133.3 MHz |
| $b_{1a_2}$ | 3 |
| $b_{1a_3}$ | 2 |
| $b_{1a_4}$ | 12 |
| $b_{1a_5}$ | 2 |
| $b_{1a_6}$ | 2 |
| $f_{1b}$ | 10 MHz |
| $f_2$ | 10 MHz |
| $c$ | 4 |

Fig. 14: BLASTN: variable values in the solution.

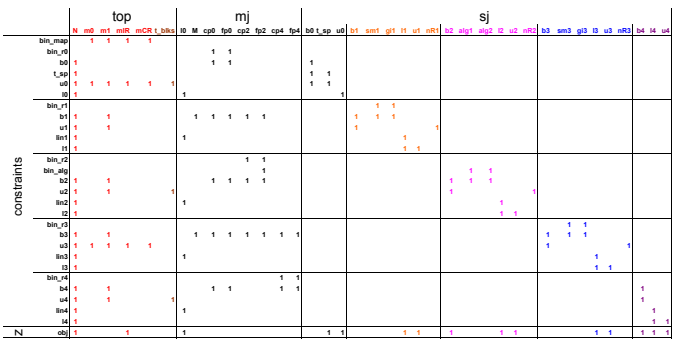


Fig. 15: Sorting variable-constraint matrix for $N = 1$.

sort blocks.

$$
\begin{aligned}
top &= \{N, m_0, m_1, m_{CR}, m_{IR}\} \\
mj &= \{\lambda_{in}, M, cpu_j, fpga_j\}, \quad for \quad j = 0, 2, 4 \\
sj &= \{smem_j, gige_j, nRes_j, alg_1, alg_2, nRes_2\}, \\
&\quad for \quad j = 1, 3 \\
mu &= \{\mu_0, \mu_1, \ldots \mu_{4N+1}\} \\
lam &= \{\lambda_0, \lambda_1, \ldots \lambda_{4N+1}\} \\
met &= \{throughput, latency\}
\end{aligned}
$$

*2) Variable-constraint matrix:* As we did with the BLASTN application, Figure 15 shows the variable-constraint matrix for the sorting application, in this case for $N = 1$. Again, each column corresponds to an individual variable (both design variables and derived variables) in the problem formulation, and each row corresponds to an individual constraint. The columns are ordered by the search procedure described in Section III-E.

For sorting, a larger fraction of the variables are complicating (in part because there exist a larger number of topology variables). There are, however, 5 Jordan blocks that correspond to each of the 5 queueing stations in this instance. (Note that this matrix is limited to the case with 2 sorts shown in Figure 5. More sorts require a larger matrix because of the larger number of variables.) These 5 Jordan blocks can be independently

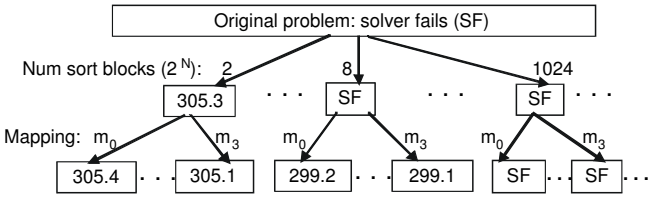


Fig. 16: Branching through $top$ variables of sort application.

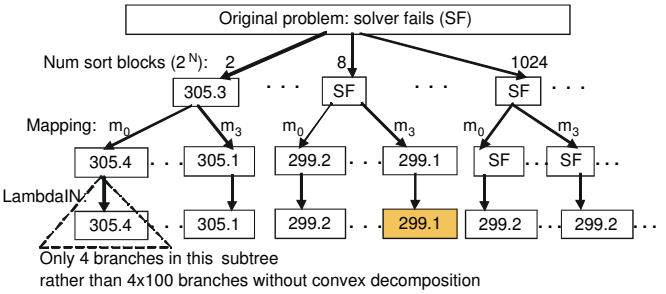


Fig. 17: Branching on $\lambda_{in}$ results in convex decomposition.

optimized, yielding significant savings in the searching of the design space.

*3) Branch and bound:* With the sorting application, none of the state-of-the-art solvers find even a feasible solution for our optimization problem formulation (based on the provided performance model presented in Section II-B) and we perform branch and bound search as we did for BLASTN. Preliminary details of the results were briefed earlier [27]. Applying our heuristic to order branching variables, we branch on the topological variables of $N$ and then mapping choices. The objective function values (equally-weighted sum of application latency and job interarrival time, in ms, to sort a batch of a million 64-bit elements) from the solutions of each of the subproblems, when the solver manages to solve the subproblem, are presented in Figure 16.

Next, we branch on $\lambda_{in}$. For the sort application, each element in $met$ moves monotonically with $\lambda_{in}$ and also each benefits from higher service rates at the different queueing stations. These properties make $z(\lambda_{in})$ convex (i.e.) $\lambda_{in}$ decomposes the problem as described in Section III-D. In other words, $\lambda_{in}$ is a convex branching variable; this means, we can solve for its value analytically rather than through branching. The reduction in the number of branch evaluations we enjoy thanks to the convex decomposition in illustrated in Figure 17, assuming that we evaluate 100 discrete values of $\lambda_{in}$ as we did with the continuous variables of BLAST.

The solver actually finds a feasible solution for every subproblem at this level and we stop branching at this level for an *anytime* solution, which is the minimum of the solutions of all the subproblems at this level (because this is a minimization problem) and is highlighted in Figure 17. The objective function value of the solution is $z = 0.3$ ms and the corresponding configuration is: 8 processing blocks, $map_{IR}$, $\lambda_{in} = 143$ 64-bit Melements/s, the fastest ($fpga_j$ instead

| Variable | Value in the solution configuration |
|---|---|
| N = 3 (num of sort blocks = $2^N$) | |
| Res. type for each compute column = FPGA | |
| Computation res. type determines communication res. type | |
| Number of resources per column | |
| $nRes_0$ | 1 |
| $nRes_1$, $nRes_2$ | 2 each |
| $nRes_3$, $nRes_4$ | 4 each |
| $nRes_5$, $nRes_6$ | 8 each |
| $nRes_7$ | N/A |
| $nRes_8$, $nRes_9$ | 4 each |
| $nRes_{10}$, $nRes_{11}$ | 2 each |
| $nRes_{12}$ | 1 |
| M = 2 (message size = $2^M$ Bytes) | |
| Bottleneck mean service rate (num. of 64-bit elems per ms) | |
| The final merge, $\mu_j$ | 10,986 |
| Input mean job arrival rate (64-bit Melements/s) | |
| $\lambda_{in}$ | 143 |

Fig. 18: Sorting: variable values in the solution.

of $cpu_j$ and $smem_j$ instead of $gige$) and maximum number of resources for every column (as allowed by $map_{IR}$), and $M = 2$ and is presented in Figure 18.

For the sorting application, the size of the search space is a strong function of the allowed range for $N$, which determines the maximum number of sort blocks. For $N$ ranging from 1 to 3 (i.e., 2 to 8 sort blocks), the size of the completely enumerated branch and bound search space is approximately $6 \times 10^{20}$ possible configurations. Using the techniques presented in this paper, this search space decreases to approximately $3 \times 10^{10}$ configurations, reducing by a factor of 20 million. As the range of $N$ increases, the size of the completely enumerated search space simply explodes, reaching $10^{260}$ for $N$ ranging from 1 to 10. At this range of $N$, our techniques yield a search space of size $5 \times 10^{12}$.

*4) Analysis of empirical results:* We verified that the solution from our framework matches the result from searching exhaustively when $N \leq 2$ (takes months beyond that). The overhead due to decomposition is reasonable because the solver runtime is in milliseconds and gets progressively lower with the subproblems. We verified the sensitivity of our solution by varying the weights on $met$.

Using our application knowledge and by neighborhood search, we found that the selection of $map_{IR}$ and $M = 2$ in the solution are not optimum even locally. To find a local optimum, we need to continue branching on the remaining $mj$ variables and, if needed, on $sj$ variables. In the particular subproblem corresponding to our solution, increasing $M = 2$ to $M = 5$ 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.

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. This is because each subproblem gets less complex progressively. After decomposing, the number of variables and constraints in any subproblem is no more than 380 and 326, respectively.

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 $(N = 3, m_{IR})$ to $(N = 4, m_0)$ with a corresponding objective function value of 184.5 ms.

*5) 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 5 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 decreases the objective function value from 10 s in the initial solution down to 2 ms which is a 480-fold improvement (given that this is a minimization problem) as presented in Figure 19. In the figure, $P$ is the original problem that evaluates all values of $N$ and all the mapping choices simultaneously. $P_i$ denotes 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 $P_i$ by branching 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 $P_{i\_4}$. 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.

## V. Conclusions

This paper has explored the use of queueing network models to guide the automatic design-space exploration of high performance streaming applications. The topological information about the application that is embodied in the queueing network model is used to: (1) identify Jordon blocks, (2) recognize linear chaining, and (3) discover opportunities for convex decomposition.

Two applications are used to illustrate the techniques. State-of-the-art MINLP solvers fail to find even a feasible solution for each of these applications. Both applications are solved using our heuristic for ordering branching variables with substantial savings in the resulting search space size. For the BLASTN application, the search space is reduced from approximately $10^{18}$ nodes to about $10^{12}$ nodes, representing a million-fold improvement. For the sorting application, the improvement in search space size depends strongly on the degree

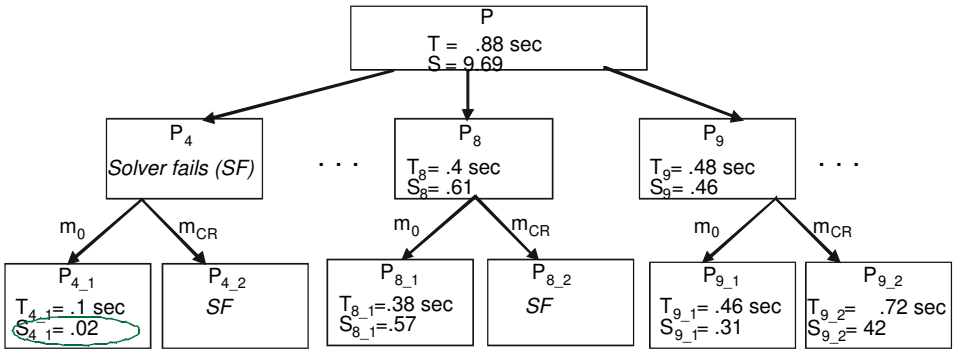


Fig. 19: Results after decomposing relaxed streaming sort.

of parallelism supported. For modest parallelism (up to eight sort blocks), the search space is reduced from approximately $6 \times 10^{20}$ nodes to about $3 \times 10^{10}$ nodes, a 20 million-fold savings. For a larger degree of parallelism (up to 1024 sort blocks), the exhaustive search space explodes to $10^{260}$ nodes while the heuristic search space is approximately $10^{12}$ nodes.

Our future investigations include: (1) Can we generalize that $top$ variables should always be considered first in branch and bound solvers? (2) Does any connected graph with the $met$ of latency and throughput facilitate our convex decomposition? (3) Can we extend the ordering heuristic to also consider potential convex branching variables and move them earlier in the ordering? (4) Investigate more rigorously the correspondence between the BCMP independence property and our decomposition techniques of linear chaining and convex decomposition.

## REFERENCES


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

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers *et al.*, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, pp. 403–10, 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman, "Gapped BLAST and PSI-BLAST: a new generation of protein database search programs," *Nucleic Acids Research*, vol. 25, pp. 3389–402, 1997.

[4] 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*, vol. 22, no. 2, pp. 248–260, 1975.

[5] D. Bertsekas, *Nonlinear Programming*, 2nd ed. Athena Scientific, 2003.

[6] J. D. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. Chamberlain, "Mercury BLASTN: Faster DNA Sequence Comparison Using a Streaming Hardware Architecture," in *Reconfigurable Systems Summer Institute*, Jul. 2007.

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

[8] R. D. Chamberlain, M. A. Franklin, E. J. Tyson, J. H. Buckley, J. Buhler, G. Galloway, S. Gayen, M. Hall, E. B. Shands, and N. Singla, "Auto-Pipe: Streaming applications on architecturally diverse systems," *Computer*, vol. 43, no. 3, pp. 42–49, Mar. 2010.

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

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

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

[12] 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*, 2009, pp. 201–208.

[13] W. J. Dally *et al.*, "Merrimac: Supercomputing with Streams," in *ACM/IEEE Supercomputing Conf.*, 2003.

[14] A. E. Darling *et al.*, "The design, implementation, and evaluation of mpiBLAST," in *4th Int'l Conf. on Linux Clusters*, 2003.

[15] A. Dasgupta and R. Karri, "Optimal Algorithms for Synthesis of Reliable Application-Specific Heterogeneous Multiprocessors," *IEEE Transactions on Reliability*, vol. 44, no. 4, pp. 603–613, Dec. 1995.

[16] S. Datta, P. Beeraka, and R. Sass, "RCBLASTn: Implementation and evaluation of the BLASTn scan function," in *Proc. of 16th Int'l Symp. on Field-Programmable Custom Computing Machines*, Apr. 2009.

[17] R. Dor, J. M. Lancaster, M. A. Franklin, J. Buhler, and R. D. Chamberlain, "Using queuing theory to model streaming applications," in *Proc. of Symp. on Application Accelerators in High Performance Computing*, Jul. 2010.

[18] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: high performance graphics co-processor sorting for large database management," in *Proc. of SIGMOD Int'l Conf. on Management of Data*, 2006, pp. 325–336.

[19] C. Grozea, Z. Bankovic, and P. Lasko, "FPGA vs. multi-core CPUs vs. GPUs: Hands-on experience with sorting," in *Facing the Multi-Core Challenge: Conference for Young Scientists at the Heidelberger Akademie der Wissenschaften*, 2010.

[20] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, "Single pass streaming BLAST on FPGAs," *Parallel Computing*, vol. 33, pp. 741–756, 2007.

[21] 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.*, vol. 4, no. 4, pp. 1–34, 2008.

[22] 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.

[23] B. C. Lee and D. Brooks, "Roughness of microarchitectural design topologies and its implications for optimization," in *High Performance Computer Architecture*, 2008, pp. 240–251.

[24] H. Lin, X. Ma, P. Chandramohan, A. Geist, and N. Samatova, "Efficient data access for parallel BLAST," in *Proc. of 19th Int'l Parallel and Distributed Processing Symp.*, 2005.

[25] R. Marler and J. Arora, "Survey of multi-objective optimization methods for engineering," *Structural and Multidisciplinary Optimization*, vol. 26, no. 6, pp. 369–395, 2004.

[26] "Mixed Integer Nonlinearly Constrained Optimization Solvers," neos.mcs.anl.gov/neos/solvers/index.html.

[27] S. Padmanabhan, Y. Chen, and R. D. Chamberlain, "Design-space optimization for automatic acceleration of streaming applications," in *Proc.*

*of Symp. on Application Accelerators in High Performance Computing*, Jul. 2010, [Extended abstract].

[28] H. Perros and T. Altiok, "Approximate analysis of open networks of queues with blocking: Tandem configurations," *IEEE Trans. Soft. Eng.*, vol. 12, pp. 450–461, 1986.

[29] A. Pimentel, C. Erbas, and S. Polstra, "A systematic approach to exploring embedded system architectures at multiple abstraction levels," *IEEE Trans. on Computers*, vol. 55, no. 2, pp. 99–112, Feb. 2006.

[30] N. Singla, M. Hall, B. Shands, and R. D. Chamberlain, "Financial Monte Carlo Simulation on Architecturally Diverse Systems," in *Workshop on High Performance Computational Finance*, Nov. 2008.

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