A model for distributed in-network and near-edge computing with heterogeneous hardware

Ryan A. Cooke*, Suhaib A. Fahmy

School of Engineering, University of Warwick, Coventry, UK

Abstract

Applications that involve analysis of data from distributed networked data sources typically involve computation performed centrally in a datacenter or cloud environment, with some minor pre-processing potentially performed at the data sources. As these applications grow in scale, this centralised approach leads to potentially impractical bandwidth requirements and computational latencies. This has led to interest in edge computing, where processing is moved nearer to the data sources, and recently, in-network computing, where processing is done as data progresses through the network. This paper presents a model for reasoning about distributed computing at the edge and in the network, with support for heterogeneous hardware and alternative software and hardware accelerator implementations. Unlike previous distributed computing models, it considers the cost of computation for compute-intensive applications, supports a variety of hardware platforms, and considers a heterogeneous network. The model is flexible and easily extensible for a range of applications and scales, and considers a variety of metrics. We use the model to explore the key factors that influence where computational capability should be placed and what platforms should be considered for distributed applications.

Keywords: Distributed computing, Edge computing, In-network computing, Hardware acceleration

1. Introduction

Distributed data processing applications involve the processing and combination of data from distributed sources to extract value, and are increasing in importance. Emerging applications such as connected autonomous vehicles rely on complex machine learning models being applied to data captured at the edge, while also involving collaboration with other vehicles. Further example applications include factory automation [1], smart grid monitoring [2], and video surveillance and tracking [3]. Such applications present a challenge to existing computational approaches that consider only the cloud and the very edge of the network. Computationally intensive algorithms must now be applied to intensive streams of data, and latency must be minimized. In these applications, data sources transmit streams of data through a network to be processed remotely, with a focus on continuous processing, and potentially involvement in a feedback loop, as opposed to other applications that involve large scale storage and delayed processing. Latency, the time taken to extract relevant information from the data streams, and throughput, the rate at which these streams can be processed, are key performance metrics for such applications.

Centralized cloud computing is often utilized in these scenarios, since the data sources do not typically have adequate computing resources to perform complex computations. Applications also rely on the fusion of data from multiple sources, so centralized procesing is useful. The cloud also offers benefits in scalability and cost, and has been shown to provide benefits in applications such as smart grid processing [2, 4] and urban traffic management [5].

However, many emerging streaming applications have strict latency constraints, and moving data to the cloud incurs substantial delay. Furthermore, while the data generated by sources can be small, a high number of sources means that, in aggregate, the volume of data to be transmitted is high. For example, in 2011, the Los Angeles smart grid required 2TB of streamed data from 1.4 million consumers to be processed per day [2]. Some applications, such as those dealing with video data, must also contend with high bandwidth data requirements.

These limitations have led to an increased interest in 'edge' or 'fog' computing, a loosely defined paradigm where processing is done either at or close to the data sources. This could mean at the source, such as on a sensor node with additional processing resources [6]. It can also encompass performing processing within the network infrastructure, such as in smart gateways [7], or in network switches or routers. Cisco offer a framework that allows application code to be run on spare computing resources in some network elements, and Ethernet switches from Juniper allow application compute to be closely coupled with the switching fabric.

Edge computing can also include the concept of 'cloudlets',

⋆This work was supported in part by The Alan Turing Institute under the UK EPSRC grant EP/N510129/1.

*Corresponding author

Email addresses: ryan.cooke@warwick.ac.uk (Ryan A. Cooke), s.fahmy@warwick.ac.uk (Suhaib A. Fahmy)

which are dedicated computing server resources placed a few hops away from the data sources. These can vary in scale, from a single box placed on a factory floor to a small scale datacenter comprising multiple networked machines. While the data sources themselves may not have the required computing capabilities, these resources can support complex applications and are accessible at shorter latencies than a remote cloud [8].

In complex applications, it is likely that some processing, such as filtering and pre-processing can be performed at the edge, greatly reducing the volume of transmitted data, and additional processing and fusion of data can be carried out in the cloud. The benefits of this approach are that latency sensitive parts of the application can be done locally, while more computationally intensive operations that may require more processing power or additional data can be done centrally. Stream processing applications are well suited to being partitioned and distributed across multiple machines, as is common in stream processing frameworks such as Apache Storm and IBM Infosphere Streams. Additionally, cloud service providers such as Microsoft Azure have edge analytics platforms that allow processing to be split between the cloud and the edge.

Edge and in-network computing is an emerging area. Cloudlets have been utilized for image processing applications [9, 10] and augmented reality [11]. Platforms such as Google's Edge Tensor Processing Unit demonstrate that there is a trend towards moving complex computation closer to the data source. In-network computing has seen application for network functions, machine learning [12], and high data rate processing [13].

In order to explore the implications of distributing application computation across a network of heterogeneous compute platforms, a suitable model is needed. This would allow for the evaluation of different deployment strategies using metrics such as throughput and end-to-end latency. Existing models that deal with placement of processing on distributed nodes do not consider hardware resources, varied connectivity, and application features together.

To this end we have developed a generalized formulation that can represent applications and target networks with heterogeneous computing resources. It supports reasoning about in-network and near-edge processing scenarios that are emerging including both general processor based machines and hardware accelerator systems.

Figure 1 summarizes the application scenario of interest, giving an example of the type of networked system that the proposed model targets. Edge nodes such as sensors and microcontrollers transmit data through a network towards centralized computing resources. In a traditional cloud computing setup, only the central resources perform computation (shaded). In edge computing, the edge nodes are capable of performing some computation (shaded). In-network computing allows some tasks to be performed in the network as data traverses it, using smart switches (shaded).

The key contributions of this paper are:

- A model for evaluating different in-network computing approaches is developed, encompassing:
  - Multiple levels of network structure, unlike existing models that focus on clusters of machines.
  - Hardware heterogeneity including accelerator platforms, and the resulting differences in computing and networking.
  - Realistic representation of performance metrics, alongside energy and financial cost.
- The model is used to examine a case-study scenario and draw general lessons about in-network computing on different platforms using a set of synthetic applications.

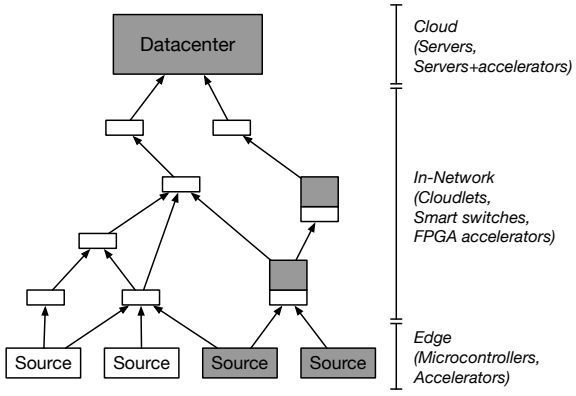


Figure 1: An example of the type of networked system that the proposed model targets. Shaded nodes can perform computation.

2. Related Work

Distributed Stream Processing Models: The allocation of streaming tasks to networked processing nodes has been explored in a variety of existing work. Applications are represented as a graph of tasks with edges representing dependencies, while networks are represented as a graph of compute nodes with edges representing links.

Earlier models such as Aurora/Medusa [14] focused on load balancing in task placement, primarily for the allocation of tasks to multiple servers in a datacenter environment. However, network costs are not modelled, making them unsuitable for scenarios that consider larger scale networks where communication and network costs are more significant. Work on more network-aware placement [15, 16, 17, 18, 19] was tailored towards networks of machines that are more widely distributed, and include network utilization and latency in their formulation. These models are all focused on placing operators to optimize specific objectives, for example bandwidth utilisation, meaning that they aren't generalisable when wanting to model a range of different performance metrics. Since these online optimisations are run dynamically, the models are significantly simplified to minimize their impact on the application. These models consider homogeneous processor platforms and do not support alternative hardware platforms with different computational models and metrics.

Recently, more generalized placement models have emerged [20, 21, 22]. These focus on creating a general representation of the operator placement problem, developing formulations based on

integer linear programming instead of focused heuristics. They are still limited as they assume a fully connected cluster of machines, and their models of computing resources and tasks are coarse grained. We are interested in a scenario where hardware acceleration may be utilized at certain computing nodes, using a different computational model to that of a processor, which is considered solely in these models.

In contrast to these works, our proposed model is not focused on finding an optimal allocation of tasks to a given set of resources at runtime. Instead, we wish to use it to investigate the implications of placing computing resources at different locations in a network and to understand the benefits and costs of doing so. Since we are not concerned with dynamic optimisation of operator placement within a time constraint, the model can include more fine grained detail for tasks and hardware, accounting for hardware acceleration, heterogeneous resources required by tasks, the financial cost of adding additional compute capability to network nodes, and energy consumption. We also consider the networked system as a whole, from the sensor nodes to the datacenter, instead of focusing on a cluster of computational servers. The focus of this paper is not optimisation, but rather an analysis of different distributed computing paradigms in the context of streaming applications.

Edge/Fog Computing: In response to increasing demand for low latency in distributed streaming applications, efforts have been made to move computation closer to the data source, or the 'edge' of the network. Where processing occurs varies, and it is rare that the application is entirely pushed to the edge. Typically operations such pre-processing and filtering take place at the edge, with aggregation and decision making centralized. This approach has been applied to domains such as smart grid, radio access networks, and urban traffic processing [23, 24, 25]. The model we have developed is capable of representing this scenario.

In some cases a majority of the processing is performed at the data source. This is common in sensor networks, where communication costs are higher than computation costs. Some examples of this are TAG [26], directed diffusion [27], EADAT [28], and MERIG [29]. These models consider computing at the very edge of the network, unlike those discussed previously. Our proposed model can account for the energy costs of communication and computation, as well as representing heterogeneous network links, unlike these models.

Processing may also be offloaded to local 'cloudlets', servers dedicated to computation, a few hops away in the network from the data source. This approach can be seen in mobile edge computing, where processing data on a mobile device would consume too much power, and doing so in the cloud would lead to high latency [30]. Cloudlets have also been demonstrated in video processing and augmented reality applications [9, 10, 11] where latency is an important consideration.

In-network computing is another emerging paradigm in which traditionally centralized computation is distributed throughout the networking infrastructure. Devices such as network switches and gateways are extended to perform additional data processing as well as their network functions. This technique has been demonstrated to result in a reduction in data and execution latency in map reduce applications [12]. A key value store implemented on an FPGA based NIC and network switch outperformed a server based implementation [13]. In-network computation using programmable network switches for a consensus protocol was demonstrated in [31]. As the capability of this hardware improves, this methodm in which networking elements are used for both moving data as well computing, is becoming more viable. Extending such capabilities to broader applications requires the ability to analyse applications composed of multiple dependent tasks and determining how to allocate these to capable nodes. Our proposed model allows this to be explored in a manner not possible using existing distributed computing models.

Hardware acceleration: A primary motivation for this work is the increasing complexity of applications, growing volumes of data, and more widespread availability of alternative hardware such as GPUs and FPGAs that can boost the performance of these applications. Recent work has explored accelerators for a variety of algorithms relevant to networked systems [32, 33, 34]. Within the datacenter, heterogeneity has emerged as an important way to address stalled performance scaling and rising energy constraints. FPGAs can be integrated into datacenter servers for application acceleration [35]. FPGA partial reconfiguration [36] allows these hardware platforms to support sharing and virtualisation of multiple accelerators that can be changed at runtime [37], hence offering some of the flexibility of software platforms with the connectivity and performance of hardware. This trend is expected to continue with the deployment of FPGAs in commercial cloud computing datacenters [38, 39]. Tightly coupling accelerators with the network interface has also been demonstrated to be effective in embedded networks [40] and the datacenter [41], and to have significant impact on streaming application latency [42]. To reflect the trend towards heterogeneity, our proposed model encompasses the idea of distinct hardware platforms with different computational characteristics. This further differentiates our work from others that consider only traditional processor based compute architectures.

3. Scenario and Metrics

The scenario of interest comprises a set of distributed data sources producing continuous streams of data, connected through a network comprised of intermediate nodes (for example gateways, routers, or cluster heads) to a central data sink, such as a datacenter. These data sources could be cameras, streams of documents, environmental/industrial sensors, or similar. An application consisting of a set of tasks and their dependencies processes these streams to make a decision or extract value. These tasks operate on the different streams of data, and some combine information from multiple (possibly processed) streams. Individual tasks affect the data volume through a reduction factor that determines the ratio of input data to output data, which reflects the properties of many stream processing tasks. An example of such an application is a smart surveillance system that monitors video streams from many cameras to detect specific events. Video streams can come from a mix of fixed cameras

and mobile platforms, with different resolutions, frame-rates, and interfaces, requiring different amounts of processing. The application uses processed information to adapt how the cameras are deployed and positioned.

In order to evaluate alternative allocations of resources and tasks, we consider the following key metrics of interest, with some explanation of how they are impacted below. We provide the comprehensive formulation of these metrics in Section 5.

3.1. Latency

Latency is important when data is time-sensitive. Fast detection of an event may have safety or security implications, or in some applications, there could be real-time constraints. In this case-study, transmitting all video streams to the cloud introduces large communication delays and competition for resources in the cloud can add further latency. Performing computation closer to the cameras, whether at the cameras or in network switches can reduce these communication delays, and distributing the tasks to different network nodes reduces the delays from sharing centralized resources. Even with less powerful hardware, latency can improve as a result of this stream processing parallelisation.

3.2. Bandwidth

Processing sensor data often reduces the size of data, outputting filtered or aggregated data, or simple class labels. Hence, if this processing is performed nearer to the data source, bandwidth consumption further up the network can be reduced significantly. There may also be scenarios where early processing can determine that a particular stream of data is useless, and hence further transmission can be avoided. In our example, some cameras may use low resolutions or frame rates, and hence be less costly in terms of bandwidth, while others might require significantly higher bandwidth, which would be more efficiently processed nearer to the cameras. It is clear once again that this decision depends on the specific application and tasks.

3.3. Energy

Energy remains a key concern as cloud computing continues to grow; the power consumption of datacenter servers and the network infrastructure required to support them is significant. One approach vendors have taken to try and address this is to introduce heterogeneous computing resources, such as FPGAs, to help accelerate more complex applications while consuming less energy. However, these resources add some energy cost to the datacenter, in the hope that this will be offset by significantly increased computational capacity. There is similarly an energy cost for adding accelerators in the network infrastructure but this is likely less than the cost of full server nodes, and leads to a reduced load on the datacenters as they then only deal with processed data. However, it is clear that energy consumption is heavily dependent on where such resources are placed. It is also possible that energy constraints at source nodes can impact what can be done there. In this example, battery-powered drones carrying cameras may have constrained power, so performing more computing there may not be viable.

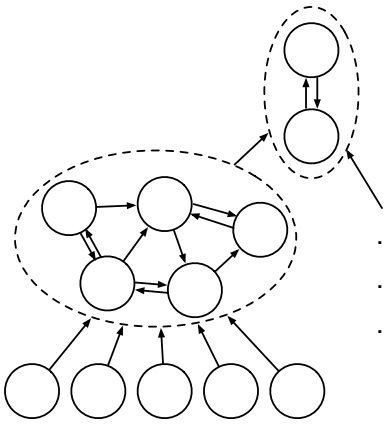


Figure 2: Nodes in the network graph can represent a single device or a cluster of networked devices.

3.4. Financial Cost

Adding computing capabilities to all data sources is expensive, especially where the tasks to be performed are computationally expensive, possibly requiring dedicated hardware. In this example, the cameras would have to be smart cameras with dedicated processing resources attached, and this is likely to increase cost significantly. While centralising all computation is likely to be the cheapest solution in terms of hardware, placing some computation in the network can come close to that cost, while offering significant benefits in the other metrics.

4. Proposed Model

The proposed model defines a network topology, task/operator graph, and hardware platforms. Tasks and hardware platforms can be allocated to network nodes, and values for the previously mentioned performance metrics can be calculated. The network communication topology is assumed to be pre-determined, though not the hardware at the nodes or the task allocation. The model is flexible enough to be used in a range of situations.

The logical topology of the network is represented as a graph, $G_N = (N, E_N)$, where N is the set of network nodes, with bidirectional communication across the set of edges between them, E_N. Application data travels through these nodes and edges towards a central sink. A node can represent either a single machine in the network, such as a gateway, switch or, server, or a 'tier' or 'level' of the network infrastructure. In this case a node represents multiple machines but the connectivity between them is not modelled at the higher level in the graph (see Figure 2). Using this representation allows the network topology to be represented in a tree structure as suits the application models considered.

To represent the application, a directed acyclic graph (DAG) is used to define the relationships between tasks, $G_T = (T, E_T)$, where T is the set of tasks and E_T defines the dependencies between them. G_T is a tree structure with a global task at the root, with other nodes representing sub-tasks, such as aggregations and pre-processing. This task model is based on the stream processing model, where data is processed per sample as it arrives.

Each task $t \in T$ can be assigned to a node through an *implementation*. Implementations are pieces of software or hardware logic that can perform the required task. This allows for selection between software implementations and hardware architectures that may have different benefits and drawbacks. This is in contrast to previous work which typically considers generic query operators and does not allow for the possibility of alternative implementations of a task. Implementations and tasks are treated as black boxes that take inputs and produce outputs, and have already been benchmarked to determine an estimate of processing time and energy consumption on a reference platform with no other tasks running. A set of platforms that can be assigned to nodes, P, to execute the tasks can have varying computational models and available resources.

4.1. Tasks

$T = \{t_1, t_2, t_3 ... t_T\}$ is the set of application tasks to be allocated to nodes in the network. Individual tasks represent functions to be carried out on a data stream. Together tasks represent the operations performed on each data stream, and specify how they are combined and manipulated to extract value. In this model, data is consumed by a task and transformed, with the result passed to the parent task. Task dependency is captured in the DAG, with each task unable to begin until all of its child tasks have been completed on a given instance of data—tasks with multiple children are typically aggregation operations. Each task $t \in T$ is defined by $t = (f_t, M_t, C_t, a_t)$.

- The set $C_t \subset T$ contains the prerequisite tasks for t that must be completed before task t can begin—its child tasks;
- $a_t \in T$ is the parent task of t, which cannot begin until t has finished.
- f_t is the reduction factor, where $0 < f_t \leq 1$. This parameter represents the amount that a task will reduce the volume of data it operates on;
- $M_t \subset M$ is the set of implementations that can implement the functionality of t;
- The data into (operated on by) a task t, denoted δ_t, is the sum of the data out from all sub tasks, $\delta_t = (\sum_{i=0}^{|C_t|} d_i)$;
- The data output from a task, d_t, to be processed by the task's parent task, is given by $d_t = f_t \delta_t$.

This representation of tasks supports different types of operations, for example, a filtering tasks that reduces a data stream, or aggregation tasks that merge multiple streams. Traditionally, aggregation tasks that process several data streams would have to be centralized but in this model they can be placed at intermediate nodes that has access to the requisite streams.

4.2. Implementations

$M = \{m_1, m_2, m_3 ... m_M\}$ is the set of all implementations, which are the pieces of software or hardware that implement the functionality of a task. Implementations can represent different software algorithms or hardware accelerator architectures that give the same functionality but have different computational delays or hardware requirements. Each task $t \in T$ has a set of implementations M_t, and each $m \in M$ is defined by $m = (t_m, \tau_m, R_m, h_m)$

- $t_m \in T$ is the task that is implemented by m;
- the set $R_m = \{r_{m1}, r_{m2}, r_{m3} ... r_{R_m}\}$ contains the amount of each resource needed to be able to host the implementation, such as memory, FPGA accelerator slots, etc;
- τ_m is the time taken for this implementation to complete the task it implements per unit of data, compared to a reference processor;
- $h_m = \{0, 1\}$ signals whether the implementation is software or hardware. A value of 0 is software, 1 is hardware.

4.3. Platforms

Platforms represent the systems in a network node that can carry out tasks. We define $P = \{p_1, p_2, p_3 ... p_P\}$ as the set of platforms that could be assigned to node $n \in N$. Each platform $p \in P$ is defined by $p = (e_p, c_p, w_p, R_p, h_p)$, where:

- e_p is the execution speed of the platform relative to a reference processor—this represents different processors having different computing capabilities;
- c_p is the monetary cost of the platform;
- w_p is the power consumption of the platform;
- $R_p = \{r_{p1}, r_{p2}, r_{p3} ... r_{pR}\}$ is the set of resources available on the platform, such as memory, FPGA accelerator slots, etc. Resources are required by implementations;
- $h_p = \{0, 1\}$ indicates whether the platform runs software or hardware versions of tasks. A value of 0 means the platform is a processor that executes software, and a value of 1 means the platform is a hardware accelerator that executes application-specific logic. This is used to ensure correct allocation of software and hardware implementations.

Unlike existing work, this model makes the distinction between platforms that execute software code and hardware acceleration platforms such as FPGAs as they have different computational delay models, discussed in Section 5.1. Hardware acceleration platforms incur no latency penalty when multiple tasks are present on the same node, whereas software platforms do, as a result of contention for computing resources.

4.4. Network

$N = \{n_1, n_2, n_3 ... n_N\}$ is a set of the network nodes, for example sensors, gateways, and routers, or servers. Each $n \in N$ is defined by $n = (a_n, C_n, P_n, b_n)$, where:

- $a_n \in N$ is the parent node of n linking it to towards the central data sink;

- $C_n \in N$ is a set of child nodes of n linking it to towards the source(s);
- $P_n \subset P$ is the set of platforms that can be assigned to node n. For example, a large datacenter class processor that must be housed in a server rack cannot be placed on a drone;
- b_n is the outgoing interface between the node n and its parent node, and represents the bandwidth in terms of data per unit time.

4.5. Sources and Data

$S = \{s_1, s_2, s_3 ... s_S\}$ is the set of data sources. We model data as continuous streams, as we are interested in applications that process and merge continuous streams of data. A data source could represent a sensor, database, video, or other source that injects a stream of data into the network. Each $s \in S$ is defined by $s = (n_s, t_s, d_s, e_s)$.

- $n_s \in N$ is the parent node of the source, the node where the data stream enters the network;
- $t_s \in T$ is the task to be performed on data being produced by the source;
- d_s is the amount of data in one instance from this source per period e_s;
- e_s is the period between subsequent units of data of size d_s entering the network.

The model assumes a constant periodic stream of data from the source, such as a periodic sensor reading, frame of a video, or set of captured tweets for example. There are some systems that do not fit this model – for example where sensors may only send out data if there is some change detected. This case can still be represented in the proposed model, as the sensor is still continually capturing data as a source and the detection component can be modelled as a filtering task that reduces it.

4.6. Allocation Variables

Boolean variables represent the allocations of tasks and hardware to network nodes. $x_{nm} = \{0, 1\}$ represents the allocation of an implementation $m \in M$ to node $n \in N$. Similarly, $y_{np} = \{0, 1\}$ represents the allocation of platform $p \in P$ to node $n \in N$. $z_{nmp} = \{0, 1\}$ represents the allocation of platform $p \in P$, and task $m \in M$ to a node $n \in N$, using a set of constraints.

A summary of the symbols used in the model is presented in Table 1.

4.7. Constraints

Constraints are used to ensure correct allocation of tasks, platforms, and nodes.

4.7.1. Allocate tasks only once

$$\forall t \in T, \sum_{i=0}^{|N|} \sum_{j=0}^{|M_t|} x_{ij} == 1 \tag{1}$$

Symbol	Meaning
x_{nm}	allocation of implementation m to node n
y_{np}	allocation of platform p to node n
z_{nmp}	allocation of m and p to n
$u_{nm_1m_2p}$	allocation of m_1, m_2, and p to n
τ_{max}	maximum path delay
g	throughput
$K_t \subset T$	set of tasks lower than t in task sub-tree with t at the root
$K_n \subset N$	set of nodes lower than n in network sub-tree with n at the root
$D_s \subset N$	set of nodes on path from s to root node
v_{np}	1 if $p \in P_n$, 0 otherwise
$P_h \subset P$	set of all platforms that run hardware implementations
$P_s \subset P$	set of all platforms that run software implementations
H	set of all paths from leaves to root in task graph
$H_t \subset H$	set of tasks on path from leaf task t to root
O_{H_t}	Set of all other tasks not on path H_t
$I \subset M$	Set of all software implementations
ϕ_{mpt}	Time to complete task implementation on node
q	Bandwidth of streams / tasks
$L \subset T$	Set of tasks with no child tasks
S_{K_n}	Set of all sources that lie beneath node n

Table 1: Summary of symbols used in formulation.

4.7.2. One platform per node

$$\forall n \in N, \sum_{i=0}^{|P|} y_{ni} == 1 \tag{2}$$

4.7.3. Resource availability

Allocations cannot exceed the available resources for the platform assigned to a node:

$$\forall n \in N, \forall e \in R, \sum_{i=0}^{|T|} \sum_{j=0}^{|M_i|} x_{nj} r_{je} \leq \sum_{k=0}^{|P|} y_{nk} r_{ke} \tag{3}$$

4.7.4. Additional constraints

The model allows for additional constraints to be added in order to better model a specific system or set of requirements. Constraints can be added to give certain tasks deadlines, constrain bandwidths, restrict specific nodes to certain platforms, and more.

5. Performance Metrics

As previously mentioned, there are five main metrics of interest in this analysis. Latency, throughput, bandwidth and energy consumption, and financial cost. In this section we formulate these metrics, and discuss how the formulation allows each to be evaluated.

5.1. End-to-End Latency

The end-to-end latency is the total time between an instance of data entering the network and its root task being completed. For example, this could be the time between a sensor reading

or image being taken and a fault or anomaly being detected. This value is of interest in time-sensitive applications such as those concerned with safety or closed-loop control, such as for industrial equipment, or coordinated control. The model incorporates several assumptions and behaviours that are relevant for this metric:

- Sources $s \in S$ produce continuous streams of data of an amount d_s, every period of time e_s. We take a 'snapshot' of the network at any instance of time, and say that data is entering the system at this instant from all sources, of an amount d_s. The equation we form gives the latency of the data instances entered at the beginning of this 'snapshot';
- Only one software implementation can run at a time on a node. Software runs on a first in first out basis;
- Hardware implementations of tasks operate independently from one another so can operate in parallel;
- A task cannot begin until all of its child tasks have been completed;
- Tasks start as soon as all of the data required is available, and once completed send the result to the next task as soon as possible;
- Communication and computation happen independently and can be parallel to each other;
- There is no communication time between tasks on the same node.

As tasks can only begin once their child tasks are complete, we can say the root task of the graph $G(T, E_T)$ can only start once all paths to it are complete. The end-to-end latency is therefore equal to the longest path delay of the task graph, including network and computation delay.

5.1.1. Computation Delay

The time to complete one task on the node it is allocated to can be represented:

$$\phi_{mpt} = \tau_m e_p \delta_t \tag{4}$$

For a task t, implemented with m on platform p. To find the end-to-end latency, the values of ϕ_{mpt} for each path in the task tree are summed, and the maximum value determined.

In the case of software implementations, nodes are assumed to carry out one task at a time. So in the cases of multiple tasks being assigned to the same node, in the worst case scenario, a data instance must wait for all other tasks not in the path to finish before beginning the next task. Note that this applies even if a node supports concurrent software tasks, since we assume that multiple software tasks suffer degraded performance in proportion to the parallelism applied. Unlike some other works, which are only concerned with preventing the allocated tasks exceeding a measure of available resources on a platform, running multiple software tasks at once on the same node in our model affects computational delay. For hardware implementations we make no such assumption as they can operate in parallel as separate streams since they are spatially partitioned, and so it is sufficient to only sum the path of interest, though we do factor in available hardware resources as discussed later. This distinction between software and hardware implementations of tasks better represents the growing trend of using alternative computing platforms to accelerate computation, compared with previous work that only accounts for software running on processors. Figure 3 shows this difference in scheduling for software and hardware nodes. On software nodes, tasks are performed in series in the worst case, and on hardware nodes, tasks can be performed concurrently. In this example, this means that tasks C and D can be performed in parallel to tasks A and B. Task E is dependent on tasks B, C and D, so must happen once they are completed. The added concurrency of hardware accelerator nodes helps reduce task execution latency when multiple tasks are assigned to a node.

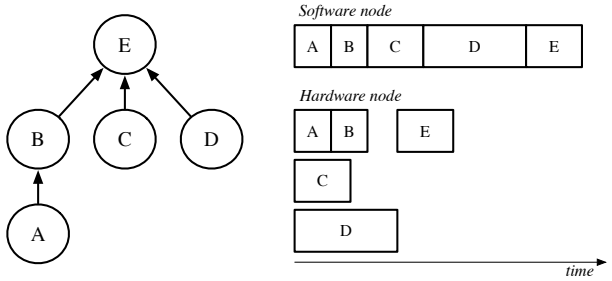


Figure 3: The difference in how a set of tasks allocated to a single node are scheduled on software and hardware accelerator nodes.

In order to represent this behaviour, a set of new allocation variables is introduced: u. Each one of these $u_{nm_1m_2p} = \{0, 1\}$ represents the allocation of two implementations m_1 and m_2 to node n, assigned platform p.

The set of tasks on the path from a leaf node on the task graph t to the root of the task graph is $H_t \subset T$. Let the set H contain all of the task path sets ($H_t \in H$). The set O_{H_t} is declared, containing all other tasks not on the path H_t. The set $I \subset M$ is defined as the set of all software implementations. The computation time for a path H_t, τ_{H_tc} in the task tree is given by:

$$\tau_{H_tc} = \sum_{i=0}^{|N|} \sum_{j=0}^{|H_t|} \sum_{k=0}^{|M_j|} \sum_{l=0}^{|P|} z_{ikl}\phi_{klj} + \sum_{m=0}^{|O_t|} \left(\sum_{q=0}^{|I_m|} u_{ikql}\phi_{qlm} - \sum_{r=0}^{|I_{a_j}|} u_{ikrl}\phi_{qlm} \right) \tag{5}$$

The z_{nmp} term in this equation is the sum of delays on all paths of the task tree. The $u_{nm_1m_2p}$ terms represent the extra delays on the path caused by having multiple tasks not on the same path allocated to the same node in software. The computation times of any other tasks allocated to the same node as any task in the path are added. The subtraction is present to ensure that this computation time is only added once for each set of tasks in a path allocated to the same node.

5.1.2. Communication Delay

A simple communication model is used where tasks send data to the parent task as soon as it is ready. There is no communication cost between tasks, only between nodes. Communication and computation can occur simultaneously and independently. If a node receives data for a task not assigned to it, it forwards this data immediately to the next node.

Data is transferred from one node to another when a task's parent task is allocated to another node. Similarly to the computational delay, we can find the total communication delay $\tau_{H_t m}$ between tasks in each path in the task tree $H_t \in H$:

$$\tau_{H_t m} = \sum_{i=0}^{|H_t|} \sum_{j=0}^{|N|} \sum_{k=0}^{|K_j|} \left(\sum_{l=0}^{|M_i|} \frac{x_{kl} d_i}{b_k} - \sum_{m=0}^{|M_{a_i}|} \frac{x_{km} d_i}{b_k} \right) \tag{6}$$

The delay from the data source s on path H_t to the node that performs the first task on it, $\tau_{H_t s}$, is given by:

$$\tau_{H_t s} = \sum_{i=0}^{|D_s|} \frac{d_s}{b_i} - \sum_{j=0}^{|K_i|} \sum_{k=0}^{|M_{t_l}|} \frac{x_{it_l} d_{s_l}}{b_i} \tag{7}$$

Where t_l is the leaf in the task path. The total communication delay in a path $\tau_{H_t k}$ is thus:

$$\tau_{H_t k} = \tau_{H_t s} + \tau_{H_t m} \tag{8}$$

The proposed model can be extended to incorporate different communication delays for software and hardware tasks as would be the case for network-attached hardware accelerators that can process packets with lower latency. The computation and communication latencies are likely to vary in reality. This model considers the worst case latency where a node processes all other tasks first and transmits the results last.

5.1.3. Total Delay

The total latency for a path, τ_{H_t}, is equal to:

$$\tau_{H_t} = \tau_{H_t k} + \tau_{H_t c} \tag{9}$$

The largest of these values is the total latency. τ_{max}.

Although we have discussed a scenario where only a single task graph is present, the model allows the possibility of multiple independent task graphs representing separate applications. Using the same method and equations, a τ_{max} can be formulated for other task graphs.

5.2. Throughput

The throughput of the system is the rate at which results are output, and dependant on the node with the longest processing time in the network. A continuous variable g can be introduced to represent the maximum delay processing stage. For software implementations, where only one task can run on a node at any time, this can be expressed:

$$\forall n \in N, g \geq \sum_{i=0}^{|T|} \sum_{j=0}^{|P_s|} \sum_{k=0}^{|M_i|} z_{nkj} \phi_{kji} \tag{10}$$

Where P_s is the set of all platforms that run software implementations. For platforms that run hardware implementations, P_h:

$$\forall t \in T, g \geq \sum_{i=0}^{|P_h|} \sum_{j=0}^{|M_t|} z_{nji} \phi_{kji} \tag{11}$$

The throughput, v, can then be expressed:

$$v = 1/max(g) \tag{12}$$

5.3. Bandwidth

Bandwidth utilization can be very significant in scenarios involving information sources with dense data and for large networks and applications. Poor utilization can also lead to additional communication delays.

The bandwidth of a data stream at a source s, q_s is given by:

$$q_s = \frac{d_s}{e_s} \tag{13}$$

The bandwidth of a task t, denoted q_t, is given by:

$$q_t = f_t \sum_{i=0}^{|C_t|} q_i \tag{14}$$

For leaf tasks t_l where $|C_t| = 0$, it is given by:

$$q_{t_l} = f_t q_s \tag{15}$$

The total bandwidth consumption at the output of a network node is the sum of the bandwidths of all streams passing through it.

$$q_{n_c} = \sum_{i=0}^{|K_n|} \sum_{j=0}^{|T|} \left(\sum_{k=0}^{|M_j|} x_{ik} q_j - \sum_{l=0}^{|C_j|} \sum_{m=0}^{|M_l|} x_{im} q_j \right) \tag{16}$$

The data not yet processed by any tasks must also be taken into account. If $S_{K_n} \subset K_n$ is the set of all sources that lie beneath n in the network graph, and $L \subset T$ is the set of all tasks where $|C_t| = 0$:

$$q_{n_l} = \sum_{i=0}^{|K_n|} \left(\sum_{j=0}^{|L|} \left(\sum_{k=0}^{|S_{K_n}|} q_k - \sum_{l=0}^{|M_j|} x_{il} q_{s_j} \right) \right) \tag{17}$$

Where q_{s_t} is the bandwidth of the source that leaf task t operates on.

The total bandwidth at a node $n \in N$ is given by:

$$q_n = q_{n_c} + q_{n_l} \tag{18}$$

This gives the bandwidth at each link between nodes.

5.4. Energy Consumption

The energy consumption of the network can be relevant for a variety of applications. In an application that deploys remote nodes with limited power sources for example, such as a wireless sensor network, energy usage can be a significant constraint. Most related works do not consider computational

energy costs. The energy used at a node $n \in N$ depends on the power consumption w_p of the platform $p \in P$ at that node, and the times taken τ_m to complete the implementations $m_t \in M$ of tasks $t \in T$ allocated to the node. Just as when formulating an equation for the end-to-end latency, taking a 'snapshot' of the network, the energy consumed by the network per data instance is given by:

$$\sum_{i=0}^{|N|} \sum_{j=0}^{|T|} \sum_{k=0}^{|M_j|} \sum_{l=0}^{|P|} z_{ikl} \phi_{klj} w_l \tag{19}$$

5.5. Financial Cost

The simplest metric is the financial cost of the solution. An equation can be formed that represents the total cost of the system, based on the platforms selected at all of the nodes. The total cost of the solution is given by:

$$c_{max} = \sum_{i=0}^{|N|} \sum_{j=0}^{|P|} y_{ij} c_j \tag{20}$$

Financial cost is a concern as it is ultimately one of the key drivers in the decision of where to place computing capability, and will always be one of the largest barriers to achieving the best possible placement. We consider the added cost of the computing platforms required to implement the in-network computation.

5.6. Combined Evaluation Metrics

We have presented formulations for the 5 important performance metrics relevant for evaluating heterogeneous distributed systems. We have kept these distinct as our proposed model is designed to be flexible enough to use for different scenarios and purposes, where the relative importance of these five metrics will vary depending on the application. Users of the model are able to build more complex metrics based on the requirements of their analysis, combining whichever of these five is relevant to their evaluation, and suitably weighting the different components.

We expect this model to be used in the design and evaluation of alternative structures for deploying heterogeneous applications. In such scenarios, a constraint-driven approach is more sensible than a combined metric, and our model supports such evaluations. For example, a required financial budget or latency target can be set and other metrics evaluated for different designs. If used to compare designs, the primary metric of importance can be evaluated, with constraints placed on the other metrics, such as the best latency for a fixed financial cost and energy budget.

We demonstrate the flexibility in this model in determining general lessons around the placement of tasks and hardware resources in our evaluation in Section 7.

6. Case Study

In this section we investigate the implications of different placement strategies in a distributed object detection and tracking system. While the formulation presented in Section 4 can be used to create an optimal placement of computing resources and tasks for a given application and network, it might be argued that such a bespoke design would not be highly practical, since a more uniform approach to deploying computing resources is generally required, and the variability of applications might make a static allocation less ideal. Hence, we evaluate strategies for a representative application to learn general lessons about the placement of computing resources in such networks. We consider a network of cameras, some fixed and some mobile, such as drones, tasked with surveying an area to detect human presence. The images collected by each camera are processed through a sequence of tasks including the histogram of oriented gradients (HOG) and an SVM classifier to detect objects of interest, and a tracking algorithm is applied that relies on the fusion of data from multiple cameras.

6.1. Network

We choose a network structure that is generally representative of that seen in an application such as this. The outermost layer represents the very edge of the network, comprising the cameras themselves (layer A). The next layer represents an access or gateway layer, that connects the cameras to the larger network (layer B). Each gateway and the connected sensors represent different areas that are to be monitored – for example rooms or neighbourhoods. Cameras connect to this layer through interfaces such as 100 Mb Ethernet or 802.11 wireless LAN. We model a transfer time of 10 ns per bit of data for this layer. The next layer is a routing layer that connects the local network to the wider network, with higher speed and bandwidth interfaces such as 10G Ethernet (layer C). Here we model a latency of 0.1 ns per bit of data. Finally, there is the cloud layer, which houses the remote computing resources. To reach this layer data must travel through the internet, for which we assume a communication time of 1 ns per bit, based on round trip times to AWS EC2 instances measured in [43]. These communication times are estimates and ignore frame/packet overheads, and many other delays, but are there to model variation in transfer time between different layers.

The topology we use in this case study is shown in Figure 4. It includes a mix of nodes with high and low fanout, and nodes at all of the layers discussed above. Links appear unidirectional as we assume data must flow through these layers in order to reach the cloud/datacenter. It is important to note that the layer B/layer C nodes do not represent individual machines, but rather layers of the network hierarchy, comprising multiple machines. Communication within these nodes is neglected in this case study.

6.2. Tasks

The HOG algorithm used in this case study has been previously implemented on a variety of computing platforms [44, 45, 46]. For the sake of this case study we break down the algorithm into 3 tasks: gradient computation, normalisation, and classification. While there are more tasks that form this algorithm, these 3 take a majority of the computation time and have a significant effect on data size. From [44] we obtain estimates

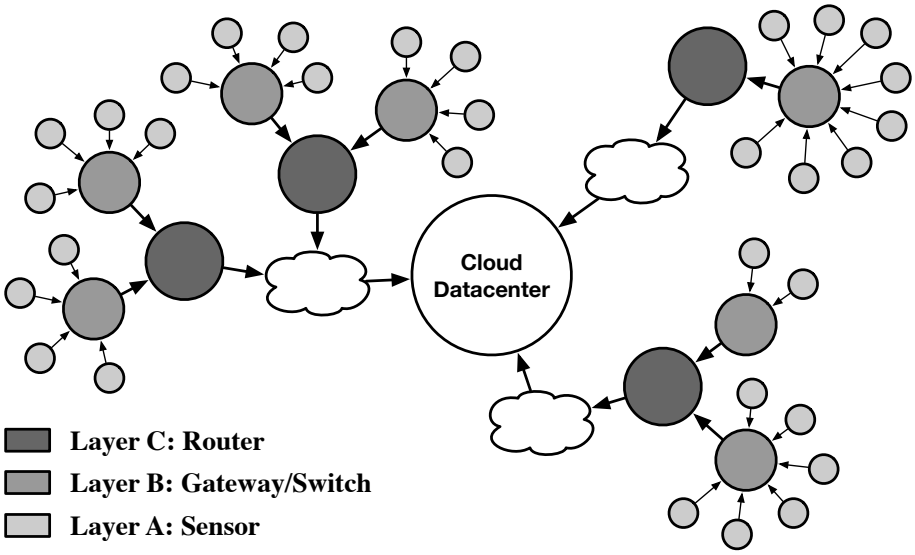


Figure 4: Network structure used in this case study.

for the reduction factor of each task. The tracking algorithm uses these HOG features and a KLT tracker [47], relying on fusion of data from multiple cameras. Therefore this task must be placed elsewhere in the network, at a location that can access all necessary cameras.

Platform	Grad,Hist	Normalisaton	Classification	Tracking
Cortex A9	2,000	3,200	1,900	2,000
Intel i7	40	60	35	40
Intel Xeon	2.6	4.0	2.3	2.6
Xilinx Zynq	260	400	240	260
Xilinx Virtex 6	1.3	2.1	1.2	1.3
Reduction factor	0.77	0.004	0.16	0.16

Table 2: Computation times in milliseconds for each task on different platforms.

In our case study, each camera has a set of tasks, *gradient comp* → *histogram* → *classification*, associated with it, and then each area of cameras has a tracking task that processes the result from multiple camera chains.

6.3. Platforms

From previous work, we estimate the computation times for each task on the different platforms. Though these are estimates, and different implementations may have varying optimisations, the relative computation times are the important factor for this case study. If computation is placed at a camera node, we assume an embedded platform. An embedded Arm Cortex A9 is used in [44] to implement the HOG algorithm, so we use the computation times presented there.

If computing is placed at the access or routing layers, we can assume a more powerful CPU is available. The work in [46] implements the algorithm on an Intel Core i7 processor. Finally, the cloud layer would use server class processors, such as the Intel Xeon platform used to implement the algorithm in [45]. We also discuss the implications of using an FPGA to accelerate tasks. The work in [44] presents an FPGA design that gives a speed up of around 7× on a Xilinx Zynq platform that could be embedded at the camera. An FPGA accelerator implemented on a larger Xilinx Virtex-6 FPGA was reported in [45], and we assume this is the FPGA platform available at other layers. We use the relative performance on these platforms to estimate the computation time of the tracker task. Table 2 summarizes time taken for each task on each platform per frame.

The costs of each platform are also relevant. In this case study we consider the extra costs associated with adding computing resources to different layers of the network. Cloud/datacenter costs are difficult to estimate, so we assume that this central node is present regardless of how we place other computing resources. Table 3 summarizes approximate costs and power consumption for each of the platforms in arbitrary currency and energy units based on costs we have determined from OEM suppliers, and manufacturer power estimation utilities.

Platform	Cost	Power Consumption
Arm Cortex A9	10	1
Intel Core i7	300	5
Intel Xeon	2000	100
Xilinx Zynq	250	5
Xilinx Virtex-6	1000	10

Table 3: Financial cost and power estimates for each platform.

The FPGA resource utilization estimates in the previously cited works suggest that both FPGA platforms can implement 3 full pipelines of the algorithm pipeline each, so 12 tasks. We assume CPU based platforms have no limit to the number of tasks that can be running, though, as discussed in the formulation, there is a latency penalty for sharing resources. We focus on latency, throughput, energy consumption, and financial cost as the metrics of interest.

We use our model to build the above scenario and evaluate different compututation placement strategies. We implement

Placement	Latency (ms)	Throughput (frames/s)	Cost	Energy
Centralised	1.95	3.43	2000	30.03
Layer C	1.97	0.88	3200	23.00
Layer B	1.93	0.94	4100	23.00
Layer A	7.16	0.14	2300	241.2

Table 4: Performance metrics for different placement strategies using software platforms.

the model in Python, using classes to represent the nodes, platforms, tasks and implementations, all containing members representing the various parameters discussed previously. Results are presented in Table 4 for software platforms and Table 5 for hardware platforms. We show the latency, throughput, financial cost, and total energy consumption of the entire system. Bandwidth results are not shown in this table as they are calculated per node in our model.

Centralized Software: A typical approach to such an application would be to centralize all tasks, performing them in software, transmitting all data to the cloud or datacenter. In this case study, this gives a latency of around 1.95 s, and a throughput of 3.4 frames per second for each camera. Note that this is in the worst case, where all camera streams compete for CPU resources. The large communication latency coupled with the large amount of data being transmitted undermines the extra computing power provided by the cloud. Energy consumption was also joint highest with this approach, as the Centralized hardware has the highest power consumption.

In-network software: An alternative approach is to push processing tasks down into the network. One possibility is placing the gradient computation, normalisation, and classification tasks on the camera nodes (layer A), and placing the tracking tasks at the appropriate layer B nodes as they require information from a set of cameras. This results in a latency of around 7.16 s and a poor throughput of 0.14 frames per second, unsuitable for real time applications. The energy consumption seems high, but this value is the energy consumption of the entire system - the consumption at each individual node is much lower. While there is communication latency, and fewer tasks competing for the same resources, the computing capability of these edge nodes is so low that the latency and throughput are much worse than the centralized placements.

Distributing tasks within the intermediate network infrastructure offers improved latency relative to placing tasks in layer A, but has minimal impact when compared to centralized placement. In this scenario, the reduced communication latency is offset by the increased computation latency. Layer B and layer C approaches introduce additional costs of 2100 and 1200 currency units respectively. The centralized solution also has 3.65× higher throughput than these approaches. This is because of its increased computing capability relative to these other nodes, meaning that there is less compututation latency. Energy consumption is less than centralized software, due to the lower power consumption of the hardware. This energy consumption is also spread across a greater number of nodes, meaning each node consumes less energy.

Centralized Hardware: Utilising FPGA acceleration at the server node reduces the latency to 1.68 s, and increases throughput to 133 frames per second, as a result of reductions in computation latency. While the FPGA should in theory provide a greater performance boost than this, the time taken for data to travel to the cloud limits the improvement that can be achieved for the application. The energy cost of running these tasks in hardware is also much lower than in software. The FPGA accelerator has a lower power consumption, as well as lower compututation time.

Placement	Latency (ms)	Throughput (frames/s)	Cost	Energy
Centralized	1.68	133	13000	1.56
Layer C	0.844	133	14000	1.56
Layer B	0.8	133	16000	1.56
Layer A	0.94	1.10	11600	30.6

Table 5: Performance metrics for different placement strategies using hardware platforms.

In-network hardware: Adding FPGA accelerators to layer C reduces latency to 0.84 s, and increases throughput to 133 frames per second due to the performance of the FPGA accelerators dramatically reducing computation latency. Placing FPGAs in layer B further improves latency to 0.83 s. These placements give improvements over the centralized FPGA approach due to the reduction in communication latency. There is little difference in latency between placing tasks predominately in layers B or C, as the fast link between these layers means that there is minimal communication delay. The disadvantage of the in-network FPGA approach is the additional cost, with the layer B and C methods costing 16000 and 14000 currency units respectively. Moving all tasks in hardware to the layer A camera nodes offers improvements over the software equivalent due to the increased computing capability. It also improves over centralized approaches due to the reduced communication latency. However the higher computation latency relative to layers B and C means that there is a higher overall latency, and worse throughput. While the total energy consumption for the layer A approach looks high, it is spread across a greater number of nodes. Each layer A node actually has a power consumption of approximately 0.956. The same processing hardware is implemented on the FPGAs in layers B and C, as well as when centralized. This results in the throughput being equal in all circumstances, despite the higher communication latency.

Optimal Placement Our model can be used with a Mixed Integer Linear Program (MILP) solver to generate a specific task and hardware placement strategy to optimize any of the performance metrics detailed in Section 5. To do this, the Python PULP front end was used to interface to the CBC solver. In this case, we optimize for latency, as in this example, energy and throughput are directly related to latency. We first generated the optimal latency placement, then ran the optimisation again with a latency constraint 5% higher than this value, but optimising for cost. This forces the solver to generate the cheap-

est placement that achieves a latency within 5% of the optimal value. As a result, our model generated a placement with the metrics shown in Table 6. This is presented for completeness; it may be argued that customising a network for a specific application is unlikely to be a common requirement. Hence, we have focused primarily on the general lessons learnt in terms of placement strategies for hardware in the network.

Placement	Latency (ms)	Throughput (frames/s)	Cost	Energy
Optimised	0.87	133	9000	1.56

Table 6: Performance metrics for MILP optimisation of model.

Summary: Improvements can be made to streaming application latency by pushing tasks into the network in either software or hardware. This also offers improvement in energy consumption at each individual node, important when there may be a limited power budget. There is a balance between the communication latency to reach higher capability nodes, and the benefits to computation latency that they provide. Placing tasks at the very edge of the network minimizes communication latency but is limited by poor computational capability. The cloud offers the highest computing capability but there is a communication latency bottleneck. The downside of using in-network task placement is the additional financial cost of the extra hardware. However, with the price/performance ratio for embedded devices scaling significantly faster than for server class CPUs, we expect this to improve over time.

6.4. Event Driven Simulation

We further developed a discrete event simulator written in Python using the SimPy library, to test the validity of results produced by our model. Data sources emit periodic packets of data into the network with the same topology and task structure. The tasks are allocated to the relevant nodes, and are executed at the nodes in a first-in first-out fashion, with priority given to the oldest data packets.

We expect differences in the reported latencies from the model and simulator primarily due to the more detailed task and communication scheduling in the simulator. The simulation processes individual packets as opposed to the considering abstract streams in the model. The data sources in the simulator emit packets with fixed periods, sources are unsynchronized, whereas the model implicitly assumes synchronisation. The simulation also takes into account a small switching delay at nodes, representing the transfer of data form received packets to the computing platform. We have not included various network related parameters in the simulation, as these are not influenced by the allocation of tasks and platforms.

Simulations of the above scenario were run for 20,000 packets entering the network from each source. The sources were fixed to the same period, but set out of sync with each other, to a degree determined from a uniformly distributed random variable. Figure 5 shows the deviation between the metrics predicted by the model and those measured in the simulation. We

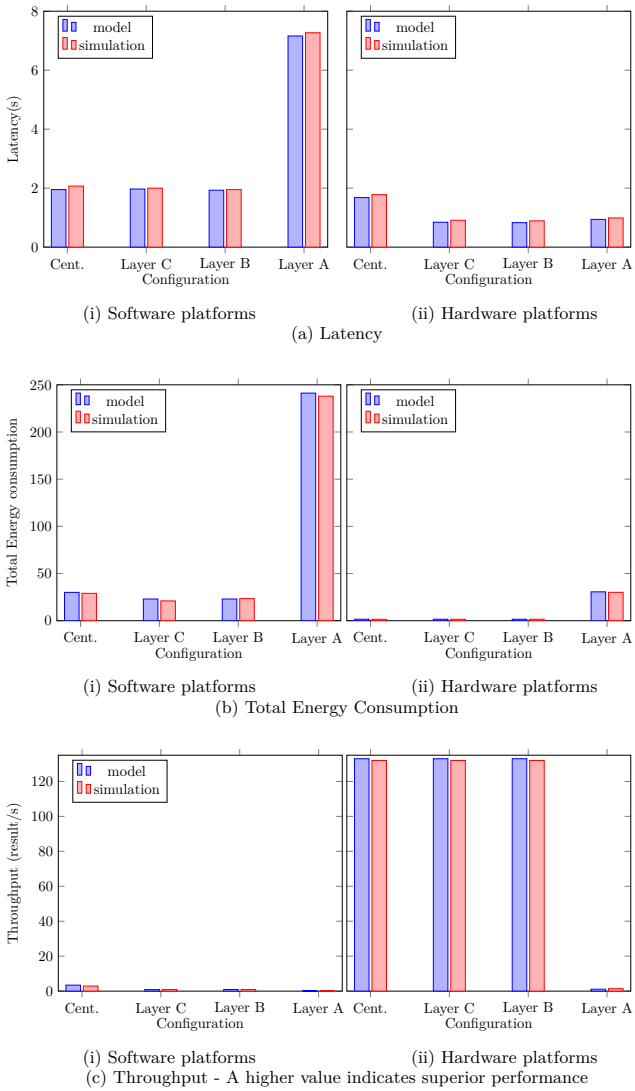


Figure 5: Difference between values calculated through the formulated model and a discrete event simulator for the same configurations and parameter values.

do not show financial cost, as there will be no difference between the simulation and model, and we do not show bandwidth as it is calculated for each individual node, not the system as a whole.

We see that if considering only software platforms, the difference between the model and simulator is close to 6%, and in hardware 7%. These differences stem from the data sources being out of sync, and the switching delays introduced at each node, not represented in the model. The ratio between computation time and network switching delay impacts this error, and hence in the case of hardware, where computation time is reduced, the overhead is more significant. However, these deviations are still well within tolerable levels.

7. Further Analysis

While determining a fixed optimal solution for a given application and network topology is possible by using an MILP

solver as we discussed, in this section, we consider synthetic scenarios in an attempt to draw general lessons about distributed, accelerated in-network computing. We explore how application and network properties influence the decision on where to place computing resources for this range of scenarios. Since latency is the primary metric of interest, we focus on that in this section.

For this analysis, we use a Python script to pseudo-randomly generate application task graphs. These are in a tree structure with a maximum depth of 4 tasks, reflect a realistic partitioning granularity rather than a very fine grained structure that would skew further towards in-network computing. We use a fixed network structure, with a similar layer A/B/C hierarchy and interface specification as used in the case study in Section 6, however with 8 layer C nodes, each serving 2 layer B nodes, each of which serves 5 layer A nodes.

Several constraints are placed on the task generation. The tree is built up leaf tasks, with a random variable determining whether each task is connected to a new task or joins one already existing in the tree. Tasks can only join other tasks whose leaf tasks originate from nodes that share the same layer B parent. We generate 100 random task trees in this manner, and the same 100 trees are used to evaluate each placement strategy, summarized in Table 7. For the purposes of this analysis, we assume that there are no restrictions on the number of tasks that can be allocated to a node, and all tasks are to be executed in 'software' - meaning that in our model there is a latency penalty dependant on the number of tasks allocated to the same node.

In this section, we focus on the latency metric, as latency reduction is one of the main motivations behind in-network and edge computing. The case study in Section 6 showed that latency and throughput are closely related for these types of streaming applications.

Strategy	**Explanation**
Centralized	All tasks allocated to root node
Pushed	All tasks pushed down toward the leaf nodes as much as possible
Intermediate	All tasks pushed down as far as possible, but not to leaf nodes
Edge/Central	Leaf tasks placed at leaf nodes, others placed centrally
Edge/Network	Leaf tasks placed at leaf nodes, others pushed down as far as possible but not to leaf nodes

Table 7: Different placement policies used in our simulations.

7.1. Relative Computing Capability

A key factor that determines where to place tasks is the relative computing capability that can be accessed at different layers of the network. In general, the closer to the centre a node is, the greater the computing capability, since the cost is amortized across more streams. The resources at the edge of the network are more likely to be limited due to space, energy, or cost constraints, while nodes further up in the hierarchy will have access to better hardware. However using better resources further up the network entails a communication latency penalty, which must be overcome by improved computation latency. For this comparison, we set tasks to have a reduction factor of 50% and equal latency on the same platform. Figure 6 shows how different placement strategies impact latency, for different relative computing capabilities.

In the unlikely case where computing capability is equal across all layers (i.e. a Centralized:B/C:A computing capability ratio of 1:1:1), pushing all tasks as close to the data sources as possible yields the lowest latency as there is minimal communication delay, and no benefit to placing tasks higher up. This may be the case if the network is a commodity cluster of homogenous machines. Computation time is also improved since the tasks are distributed across many nodes resulting in less contention than for a centralized placement.

If the compute capability at the data sources is significantly smaller (50× in this case), while the rest of the network offers equivalent computing capability (a ratio of 1:50:50), pushing tasks down to intermediate nodes offers the best latency. In this case, the slight reduction in communication latency gained through placing tasks at the data sources is outweighed by the computation latency penalty. Placing them any closer to the central node adds further communication latency with no additional benefit, and causes contention due to more tasks being allocated to fewer nodes.

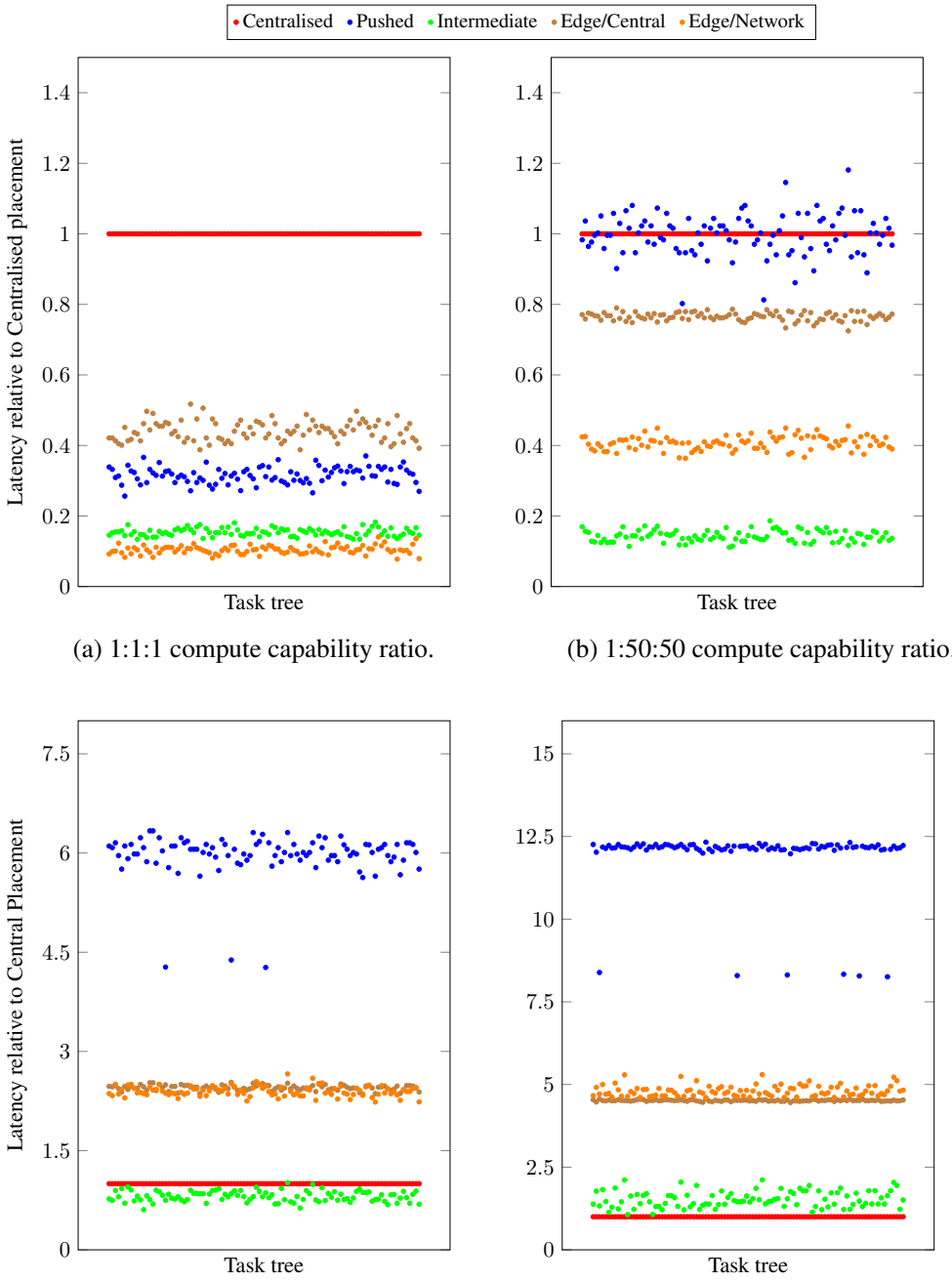
The more likely case is that resources at the central node are more capable than intermediate nodes, which offer greater capability than the edge. In the case of the central node being 5× more capable than the intermediate nodes (a computing capability ratio of 1:50:250), pushing tasks as low as possible into the intermediate nodes still outperforms the centralized solution, as tasks are distributed to a larger number of nodes, reducing computation latency. Increasing the difference in computing power to 10× (1:50:500) causes the central solution to become dominant.

Hence, we see that a key requirement for in-network computing to be feasible is that suitably capable computing resources be employed for executing tasks in the network. The more capable the edge nodes are in comparison to the root node, the greater the benefits of placing tasks further towards the edge.

7.2. Task Data Reduction

The time taken to transmit data further up the network is tied to the amount of data being transmitted. Tasks can reduce data by varying degrees, and this impacts the balance between computation and communication latency. For this experiment, we modify the reduction factors of tasks to observe the impact on latency. We use the same network topology as in the previous experiment, and the same method of generating task trees. To mimic a realistic scenario, we use a 1:50:500 relative computing capability configuration, as discussed in Section 7.1.

Figure 7 shows how different placement strategies impact latency, for different task reduction factors. If data is dramatically reduced by tasks close to the edge of the task tree, placing tasks as close as possible to the data source is more likely to provide a latency improvement as the communication cost for every other transfer between nodes is reduced. We see that



(a) 1:1:1 compute capability ratio.

(b) 1:50:50 compute capability ratio.

(c) 1:50:250 compute capability ratio.

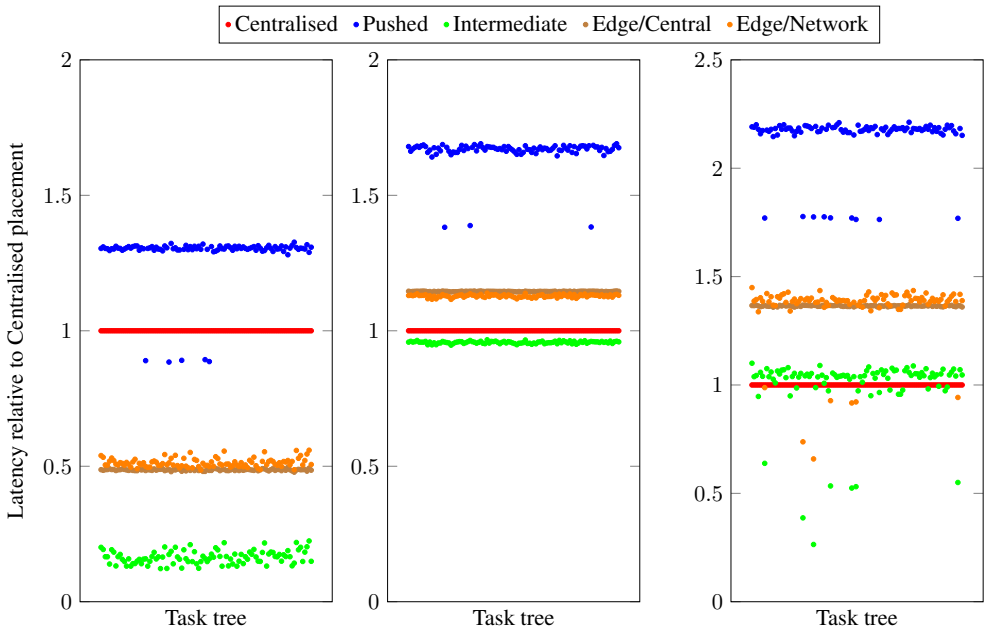
(d) 1:50:500 compute capability ratio.

Figure 6: Latency comparison for different Layer A:B:C computing capability ratios.

intermediate placement reduces latency by 5× compared to a centralized allocation in such a scenario. Placing all tasks at the edge results in 30% worse latency, despite the reduced communication latency, due to the low computing capability of these nodes. Placing only the leaf task at the edge and the rest either in the network or at the central node also provides a significant reduction in latency in this scenario.

If data is not significantly reduced in the task tree, or only at tasks higher up in the tree, placing tasks towards the central sink is preferred, especially if those resources are more capable. A centralized placement provides the best latency in a majority of cases, although only by a slight margin. For some task trees, the in-network approach is superior. This result is impacted by the relative computing capability of layers. For scenarios where the central node is much more capable than the rest of the network, the instinct is to place tasks there. However, if data is reduced significantly at the leaf tasks then placing tasks in the network can reduce communication latency significantly.

It can be seen that, generally, the closer to the edge tasks that data is reduced, the greater the benefits of placing tasks closer to the edge of the network.



(a) Significant reduction at edge tasks.

(b) Moderate reduction at edge tasks.

(c) No reduction at edge tasks.

Figure 7: Latency comparison for different edge task data reduction factors.

7.3. Network Structure

The structure of the network determines to what extent tasks can be distributed and parallelized and how much they must compete for resources. Related to this is the structure of the application task graph; having tasks that require data from multiple sources closer to the root of the tree means that tasks cannot be pushed down into the network to a layer with more computing nodes. To investigate this factor, we consider different network structures and their impact on latency, as shown in Figure 8. The tasks were generated with the same method as before, and network nodes had the same computing capability as in Section 7.2. All tasks were set to a fixed reduction factor of 0.5.

Firstly, we examine a network with low fanout, where layer B nodes each have 2 layer A nodes attached. While this means that there was more available resources towards the edge of the network, in many cases pushing tasks into the network results in almost 2× the latency of a centralized solution. Tasks that require data from more than one source must be pushed further up the network, adding additional communication latency. Additionally, as there are few layer C nodes, these nodes are over-utilized. Increasing the number of layer C nodes, or the computing capability of these nodes would offer performance benefits in this scenario.

Raising the fanout of the layer B nodes to 5 instead of 2 increases the benefits of pushing tasks into the network. As more sources share the same paths towards the central node, there is a higher chance that a task that works with data from multiple sources can be placed closer to the edge. Increasing the number of nodes at layer C in this case again slightly decreases latency, as tasks that do have to be placed there have access to more resources.

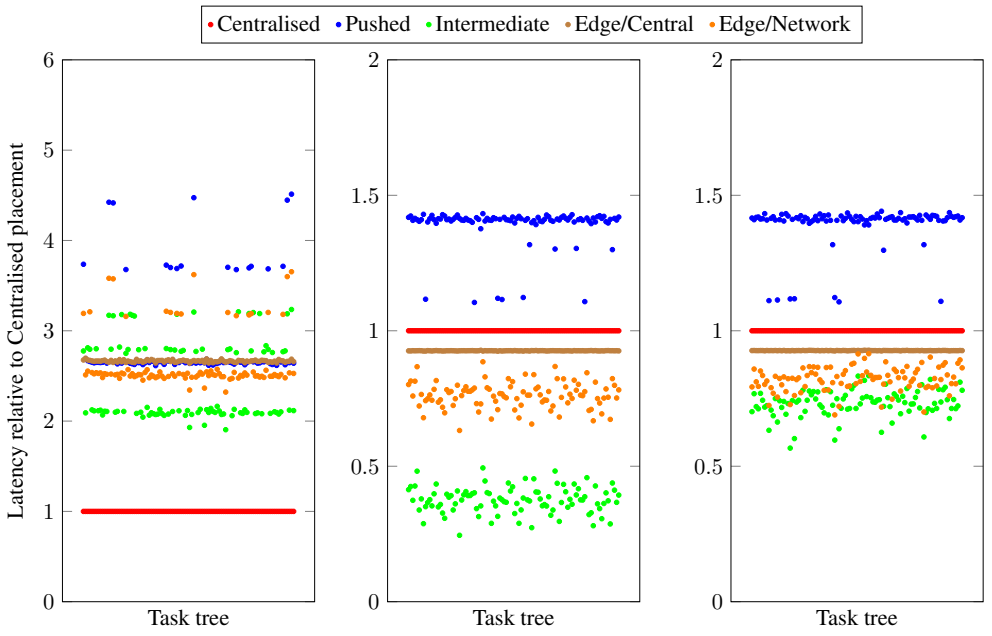
Further increasing the fanout of the layer B nodes to 20 starts to increase latency again, up to around 0.45× the centralized placement. Increasing it to 40 increases the latency to around 0.7× the central placement.

A larger fanout at layer A (the edge layer), up to a point means that there is a greater benefit of pushing tasks down towards the network edge, as there are more opportunities to place tasks that require data from multiple child tasks closer to the edge. However if the fanout is too great, resource competition starts to reduce the benefits of this approach.

It can be seen that there exists a trade-off between having multiple sources connected to the same path of nodes, and creating too much resource contention by having too many tasks assigned to the same intermediate nodes.

7.4. Hardware Acceleration

There are several key points we can take away from this investigation. In-network computing is more effective in situations where the edge and intermediate nodes are comparable in capability to the central node. While this is unlikely with traditional software processing platforms, it makes the case for trying to integrate hardware accelerators such as FPGAs into the network, as they can provide processing times closer to the more powerful processors found in a datacenter environment. We can also see that in-network computing provides more benefits in applications where data is more greatly reduced in tasks closer to the edge of the task tree. These tasks can often be large filtering or pre processing tasks, and in order to place them close the the edge of the network, more capable hardware is required. This again makes the case for hardware acceleration. Finally,



(a) Layer A fanout of 2. (b) Layer A fanout of 5. (c) Layer A fanout of 20.

Figure 8: Latency comparison for different network fanout factors.

high fanout network topologies benefit more from in-network computing as there are more opportunities for data fusion between tasks. The ability of hardware acceleration architectures to process streams of data in parallel is well suited to these scenarios, suffering less of a latency penalty due to resource contention.

8. Conclusion

The placement of computing resources and allocation of tasks in distributed streaming applications has a significant impact on application metrics. We have presented a model that can be used to reason about such applications. It models data sources that inject data into this network, applications composed of dependent tasks, and hardware platforms that can be allocated to nodes in the network. The model can be used to evaluate alternative strategies for allocating computing resources and task execution, offering information on latency, throughput, bandwidth, energy, and cost. We have used this model to demonstrate that computing in the network offers significant advantages over fully centralised and fully decentralised approaches, using an example case-study of an object detection and tracking system. We also used synthetically generated applications to explore the key application factors that impact the effectiveness of the in-network computing approach.

References

[1] N. Tapoglou, J. Mehnen, A. Vlachou, M. Doukas, N. Milas, D. Mourtzis, Cloud-based platform for optimal machining parameter selection based on function blocks and real-time monitoring, Journal of Manufacturing Science and Engineering 137 (1) (2015).

[2] B. Lohrmann, O. Kao, Processing smart meter data streams in the cloud, in: PES Innovative Smart Grid Technologies Conference Europe, 2011.

[3] W. Zhang, P. Duan, Q. Lu, X. Liu, A Realtime Framework for Video Object Detection with Storm, in: International Conference on Ubiquitous Intelligence and Computing, no. January 2017, 2014, pp. 732–737.

[4] Y. Simmhan, B. Cao, M. Giakkoupis, Adaptive rate stream processing for smart grid applications on clouds, in: International Workshop on Scientific Cloud Computing, 2011, pp. 33–37.

[5] Z. Li, C. Chen, K. Wang, Cloud computing for agent-based urban transportation systems, IEEE Intelligent Systems 26 (1) (2011) 73–79.

[6] E. Jean, R. T. Collins, A. R. Hurson, S. Sedigh, Y. Jiao, Pushing sensor network computation to the edge, in: International Conference on Wireless Communications, Networking and Mobile Computing, 2009.

[7] L. Hong, C. Cheng, S. Yan, Advanced sensor gateway based on FPGA for wireless multimedia sensor networks, in: International Conference on Electric Information and Control Engineering, 2011, pp. 1141–1146.

[8] M. Satyanarayanan, P. Bahl, R. Cáceres, N. Davies, The case for VM-based cloudlets in mobile computing, Pervasive Computing 8 (4) (2009) 14–23.

[9] T. Soyata, R. Muraleedharan, C. Funai, M. Kwon, W. Heinzelman, CloudVision: Real-time face recognition using a mobile-cloudlet-cloud acceleration architecture, in: IEEE Symposium on Computers and Communications, 2012, pp. 59–66.

[10] S. Yi, Z. Hao, Z. Qin, Q. Li, Fog computing: Platform and applications, in: Workshop on Hot Topics in Web Systems and Technologies, 2016, pp. 73–78.

[11] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, M. Satyanarayanan, Towards wearable cognitive assistance, in: International Conference on Mobile Systems, Applications, and Services, 2014, pp. 68–81.

[12] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, P. Kalnis, In-network computing is a dumb idea who's time has come, in: Proceedings of HotNets, 2017.

[13] Y. Tokusashi, H. Matsutani, N. Zilberman, LaKe: the power of in-network computing, in: Proceedings of the International Conference on Reconfigurable Computing and FPGAs, 2018.

[14] D. J. Abadi, D. Carney, M. Cetintemel, Ugur Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, S. Zdonik, Aurora: a new model and architecture for data stream management, The International Journal on Very Large Data Bases 12 (2) (2003) 120–139.

[15] Y. Ahmad, U. Cetintemel, Network-aware query processing for stream-

based applications, in: Proceedings of the International Conference on Very Large Data Bases, 2004, pp. 456–467.
[16] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, S. B. Zdonik, The design of the Borealis stream processing engine., in: Cidr, 2005, pp. 277–289.
[17] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, M. Seltzer, Network-aware operator placement for stream-processing systems, in: Proceedings of the International Conference on Data Engineering, 2006, p. 49.
[18] L. Ying, Z. Liu, D. Towsley, C. H. Xia, Distributed operator placement and data caching in large-scale sensor networks, in: Proceedings of IEEE INFOCOM, 2008, pp. 1651–1659.
[19] S. Rizou, F. Dürr, K. Rothermel, Solving the multi-operator placement problem in large-scale operator networks, in: Proceedings of the International Conference on Computer Communications and Networks, 2010.
[20] A. Benoit, H. Casanova, V. Rehn-Sonigo, Y. Robert, Resource allocation for multiple concurrent in-network stream-processing applications, Parallel Computing 37 (8) (2011) 331–348. arXiv:0903.0710.
[21] A. Benoit, H. Casanova, V. Rehn-Sonigo, Y. Robert, Resource allocation strategies for constructive in-network stream processing, International Journal of Foundations of Computer Science 22 (03) (2011) 621–638.
[22] V. Cardellini, V. Grassi, F. Lo Presti, M. Nardelli, Optimal operator placement for distributed stream processing applications, in: Proceedings of the International Conference on Distributed and Event-Based Systems, 2016, pp. 69–80.
[23] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the MCC Workshop on Mobile Cloud Computing, no. August 2012, 2012, p. 13.
[24] S. H. Park, O. Simeone, S. S. Shitz, Joint Optimization of Cloud and Edge Processing for Fog Radio Access Networks, in: IEEE Transactions on Wireless Communications, Vol. 15, 2016, pp. 7621–7632.
[25] A. Botta, W. De Donato, V. Persico, A. Pescapé, Integration of Cloud computing and Internet of Things: A survey, Future Generation Computer Systems 56 (2016) 684–700.
[26] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong, TAG: A tiny aggregation service for ad-hoc sensor networks, in: Proceedings of the Symposium on Operating systems design and implementation, Vol. 36, 2002, pp. 131–146.
[27] C. Intanagonwiwat, R. Govindan, D. Estrin, Directed diffusion, in: Proceedings of the International Conference on Mobile Computing and Networking, 2000, pp. 56–67.
[28] M. Ding, X. Cheng, G. Xue, Aggregation tree construction in sensor networks, in: Proceedings of the Vehicular Technology Conference, Vol. 4, 2003, pp. 2168–2172.
[29] H. Luo, H. Tao, H. Ma, S. K. Das, Data fusion with desired reliability in wireless sensor networks, IEEE Transactions on Parallel and Distributed Systems 23 (3) (2012) 501–513.
[30] M. Satyanarayanan, Z. Chen, K. Ha, W. Hu, W. Richter, P. Pillai, Cloudlets: at the Leading Edge of Mobile-Cloud Convergence, in: International Conference on Mobile Computing, Applications and Services, 2014, pp. 1–9.
[31] H. T. Dang, M. Canini, F. Pedone, R. Soule, Paxos made switch-y, in: Proceedings of SIGCOMM, 2016, pp. 18–24.
[32] M. Papadonikolakis, C.-S. Bouganis, A novel FPGA-based SVM classifier, in: Proceedings of the International Conference on Field-Programmable Technology, 2010, pp. 2–5.
[33] H. M. Hussain, K. Benkrid, A. T. Erdogan, H. Seker, Highly parameterized k-means clustering on FPGAs: Comparative results with GPPs and GPUs, in: Proceedings of the International Conference on Reconfigurable Computing and FPGAs, no. 1, 2011, pp. 475–480.
[34] N. Pittman, A. Forin, A. Criminisi, J. Shotton, A. Mahram, Image segmentation using hardware forest classifiers, in: Proceedings of the International Symposium on Field-Programmable Custom Computing Machines, 2013, pp. 73–80.
[35] S. A. Fahmy, K. Vipin, S. Shreejith, Virtualized FPGA accelerators for efficient cloud computing, in: International Conference on Cloud Computing Technology and Science, 2015, pp. 430–435.
[36] K. Vipin, S. A. Fahmy, FPGA dynamic and partial reconfiguration: A survey of architectures, methods, and applications, ACM Computing Surveys 51 (4) (2018) 72:1–72:39.
[37] M. Asiatici, N. George, K. Vipin, S. A. Fahmy, P. Ienne, Virtualized execution runtime for FPGA accelerators in the cloud, IEEE Access 5 (2017) 1900–1910.
[38] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J.-Y. Kim, et al., A cloud-scale acceleration architecture, in: Proceedings of the IEEE/ACM International Symposium on Microarchitecture, 2016.
[39] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung, et al., Azure accelerated networking: SmartNICs in the public cloud, in: USENIX Symposium on Networked Systems Design and Implementation, 2018, pp. 51–66.
[40] S. Shreejith, S. A. Fahmy, Smart network interfaces for advanced automotive applications, IEEE Micro 38 (2) (2018) 72–80.
[41] J. Weerasinghe, R. Polig, F. Abel, C. Hagleitner, Network-attached FPGAs for data center applications, in: Proceedings of the International Conference on Field-Programmable Technology, 2016, pp. 36–43.
[42] S. Shreejith, R. A. Cooke, S. A. Fahmy, A smart network interface approach for distributed applications on Xilinx Zynq SoCs, in: International Conference on Field Programmable Logic and Applications, 2018, pp. 189–190.
[43] O. Tomanek, P. Mulinka, L. Kencl, Multidimensional cloud latency monitoring and evaluation, Computer Networks 107 (Part 1) (2016) 104–120.
[44] M. Hatto, T. Miyajima, H. Amano, Data reduction and parallelization for human detection system, in: Proceedings of the Workshop on Synthesis And System Integration of Mixed Information Technologies, 2015, pp. 134–139.
[45] C. Blair, N. M. Robertson, D. Hume, Characterising a Heterogeneous System for Person Detection in Video using Histograms of Oriented Gradients: Power vs. Speed vs. Accuracy, Journal on Emerging and Selected Topics in Circuits and Systems 3 (2) (2013) 236–247.
[46] M. Kachouane, S. Sahki, M. Lakrouf, N. Ouadah, HOG based fast human detection, in: International Conference on Microelectronics, ICM, no. 24, 2012.
[47] B. Benfold, I. Reid, Stable multi-target tracking in real-time surveillance video, in: Proceedings of CVPR, 2011, pp. 3457–3464.

Full article citation

Ryan A. Cooke and Suhaib A. Fahmy, A model for distributed in-network and near-edge computing with heterogeneous hardware, Future Generation Computer Systems 105 (2020) 395–409.