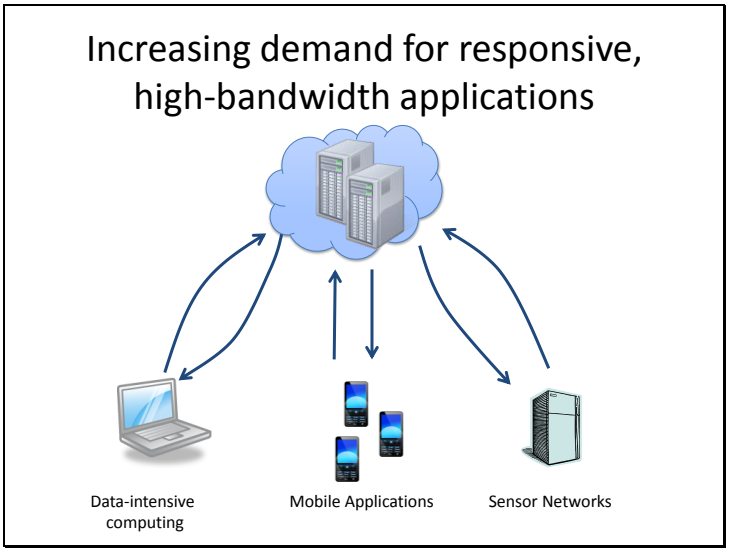
Heterogeneous Stream Computing in SAVI

ECE1548 Course Project

Charles Lo

12/4/2013

Slide 2



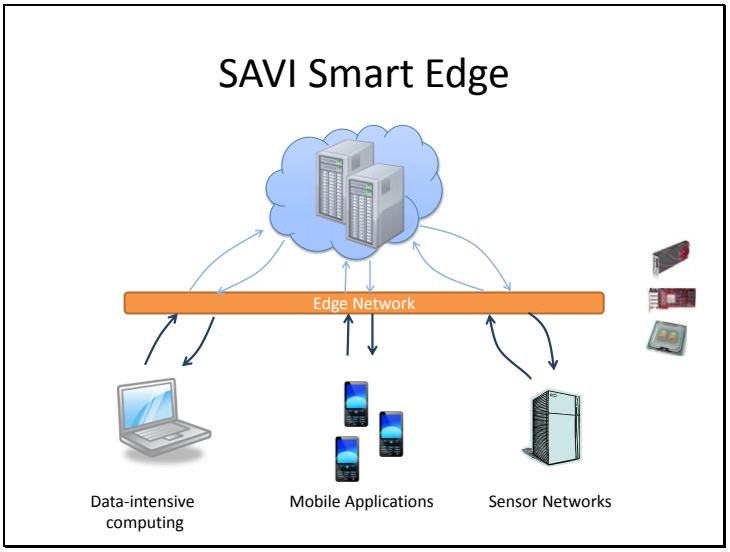
Several previous works have predicted the evolution the internet to incorporate a growing number of mobile devices as well as sensors. In turn, the demands placed on the communications network, storage and computations that serve these devices will increase.

Data from sensors such as motion detectors, temperature monitors or energy meters must be aggregated and processed. Although the data from individual sensors may be low-bandwidth, the aggregate of many sensors puts serious challenges on processing and computation. Autonomous sensors such as surveillance cameras could also require high bandwidth and their video feeds may be transcoded/compressed before they are stored or presented to a user.

Mobile users will expect consistently responsive applications while the bandwidth and processing requirements increase. For instance, high-resolution video sharing or retrieval from a growing number of users will put strain on the network.

A traditional approach with large datacenters, geographically distant from users, puts heavy requirements on the core network to ensure timely and high-capacity communication.

Slide 3

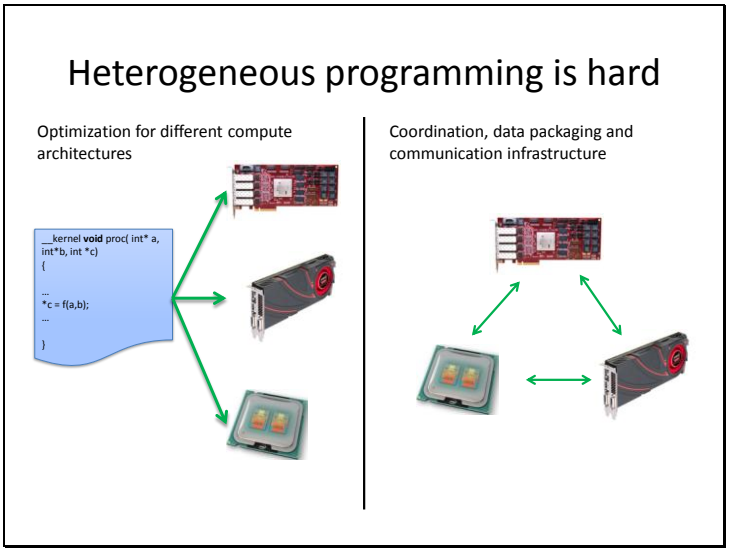


A solution explored being explored in SAVI is to use a smart edge network located at the service provider level. Such an edge contains processing and storage resources along with networking resources. Furthermore, SAVI explores the notion of virtualized resources where the capacity (in terms of communication, processing and storage) of the network can scale dynamically with demand. Such an approach provides a mechanism for energy-efficient computing where resources are only allocated as required.

By providing resources at the edge, communication between users in the same service area can be low latency leading to responsive media sharing or sensor alerts. In addition, content and time-sensitive computations can be performed at the edge. For example, real-time querying of the positions of public transit or traffic conditions could be serviced at the edge. Finally, content destined for the core can be pre-processed at the edge to reduce the load on core networks.

To support applications on the edge, SAVI integrates heterogeneous computing resources including X86 machines, field programmable gate arrays (FPGAs) and graphics processing units (GPUs) as virtual resources.

Slide 4

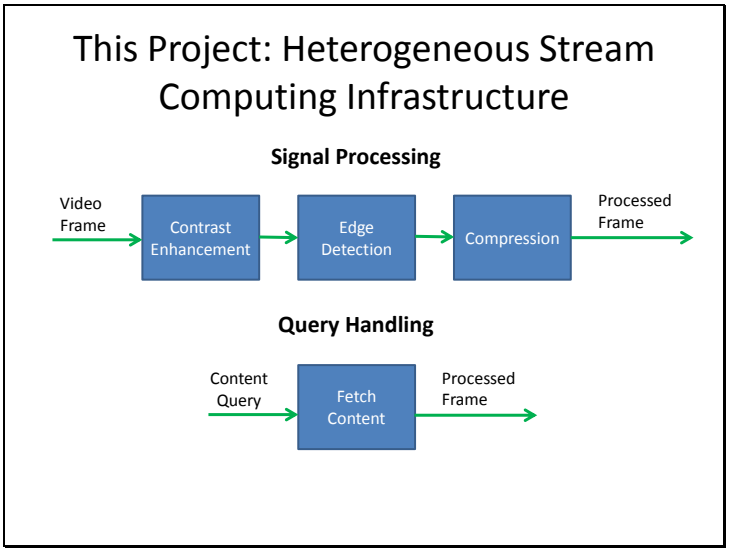


Although heterogeneous resources can be created on SAVI, developing applications on them is a challenge. In particular, two main difficulties are (1) the difficulty of designing high-performance algorithms across different architectures and (2) coordinating work across the resources.

In general, writing high-performance programs is difficult. Even on a standard X86 machine, fundamental knowledge of the architecture such as vector instructions and cache sizes are necessary to achieve maximum performance. More esoteric computing substrates such as the massively parallel, but simple, cores in GPUs or the generic digital logic units of an FPGA can be even more difficult to design with.

Once a computation has been designed for the different architectures, coordinating them is also non-trivial. For instance, simply transferring the data to operate on is hard since data structures may use pointers that have no meaning across address spaces or the endianness of words may be different. Furthermore, distributing a block of work across multiple, heterogeneous workers is challenging especially if more workers can be added dynamically. In general, work should be distributed such that the total performance is maximized.

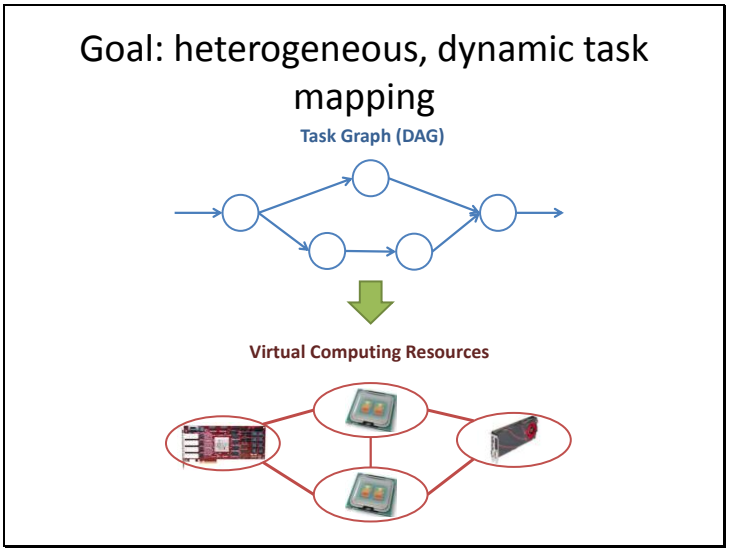
Slide 5



This project investigates a class of computations called stream computing. Stream computing is a general model where computations are defined as a series of sequential operations. Applications that fit well into this model are filtering pipelines such as for signal processing. In addition, basic content queries and responses can be modeled in stream computing.

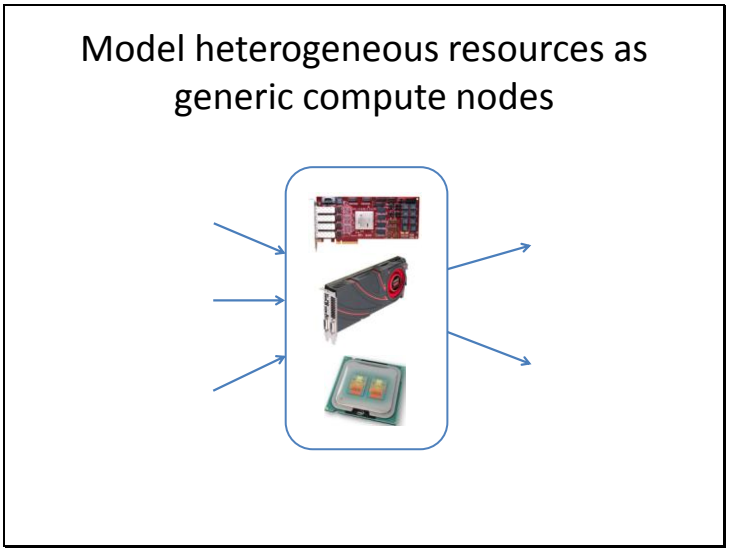
In general, applications intended for the smart edge fit well into this scheme. Examples include sensor data processing, data compression before forwarding to the core and querying databases for local information. Furthermore, FPGAs can perform streaming computations very well if the operations can be scheduled into a hardware pipeline.

Slide 7



The goal in this project is then to come up with a method of mapping a stream application, typically represented as a directed acyclic graph (DAG), onto a set of heterogeneous computing resources. The solution should facilitate mapping onto heterogeneous nodes as well as be robust to changes in the available computing resources. For instance, if a new FPGA instance is booted, the system should be able to immediately re-allocate work onto it.

Slide 9

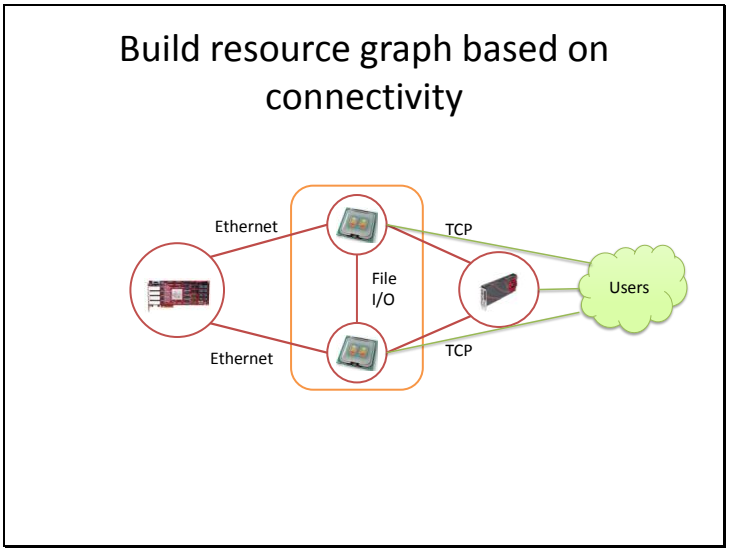


To create a framework onto which applications are mapped, we can model each heterogeneous resource as a stream computing node. The basic properties of such are a node are:

1) It can receive inputs from a number of sources
2) It performs a task based on the input data
3) It can stream the data to multiple sinks

In this model, a compute node may be a full X86 VM, a single process operating on an X86 VM, a process passing data to a GPU or a slice of FPGA resources. Notice that this model requires a generic process to run on each type of hardware. For X86 instructions this is simply a matter of compiling the compute kernel. For GPUs, the kernel must be written a little more carefully, but in general it can execute any transformation. For an FPGA, it is possible to design hardware to execute any task, but the hardware must be configured in order to perform the task. Thus, it is assumed in this model that we have a mechanism for swapping tasks on FPGA hardware, either through re-configuration or simply instantiating a new FPGA instance with the appropriate hardware to perform the new task.

Slide 10

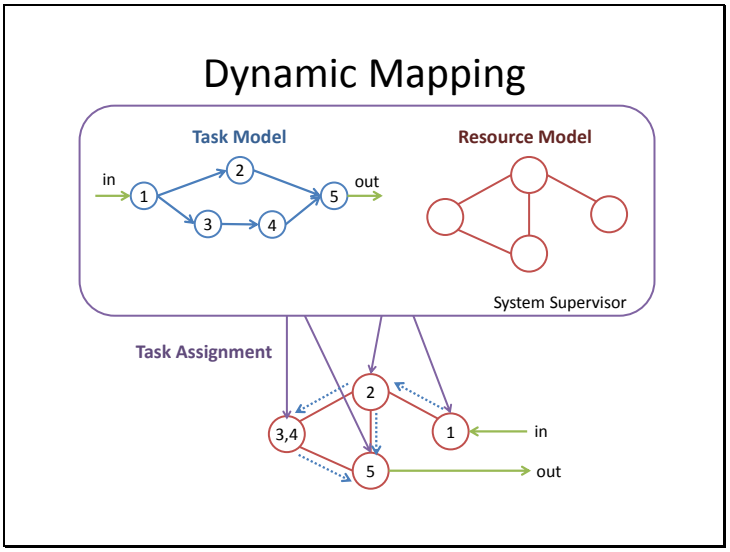


Now, given a set of virtual heterogeneous resources, we can construct a resource graph. This graph describes the connectivity of different resources and is based on the transports available. For instance, two X86 processes can be connected via File I/O or sockets. Two X86 machines may be connected via TCP or leverage a higher-level model such as a message passing infrastructure. An FPGA could be connected to other machines via raw Ethernet or possibly PCIe or even IP if the hardware is available.

This model also includes connectivity for the users such that data can enter and exit the compute resources.

The graph in general can be constructed a priori since some managing system would instantiate the nodes and so the connectivity and addresses of each resource relative to each other is known as they are created.

Slide 11



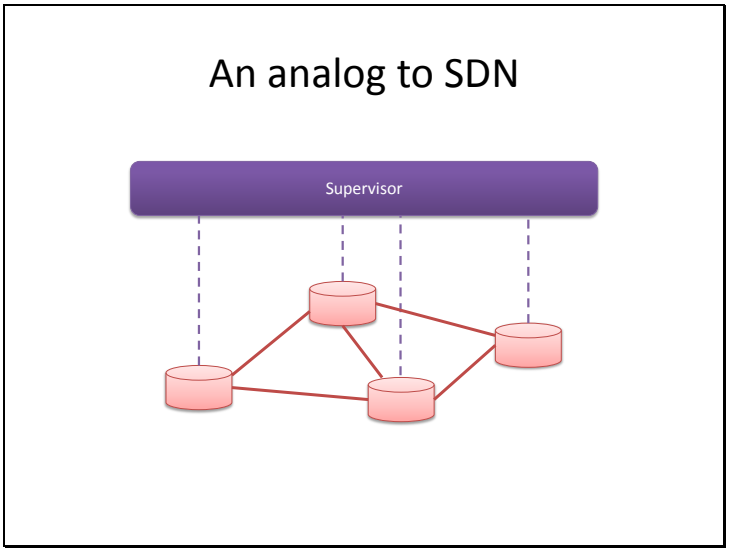
Finally, with the resource model in mind, we would like a method of distributing tasks onto them. The choice here is to use a globally-aware supervisor that examines the task graph of the application as well as the available resources and solves an optimization problem to map the tasks onto the virtual hardware.

Processes in a task graph are logically separated by the functions that must be carried out at each step. For instance, greyscale conversion followed by edge detection would be two nodes connected in the task graph. However, the actual computations could be mapped onto a single compute resource by defining a function that encapsulates both processes. In the above example, tasks 3 and 4 are both mapped onto the leftmost hardware resource as a single process.

To help the supervisor make decisions on how to map the tasks, each compute resource must have statistics or counters available that report the node's status back to the supervisor. For instance, the node could report the latency along a communication channel between neighbor nodes as well as its compute time for a particular process and the energy it would consume. Thus, the applications are mapped onto the resources based on an optimization of some objective function.

A final note is that the above image describes mapping a single application onto the resources, but the resources should be able to multitask and handle multiple applications simultaneously.

Slide 12



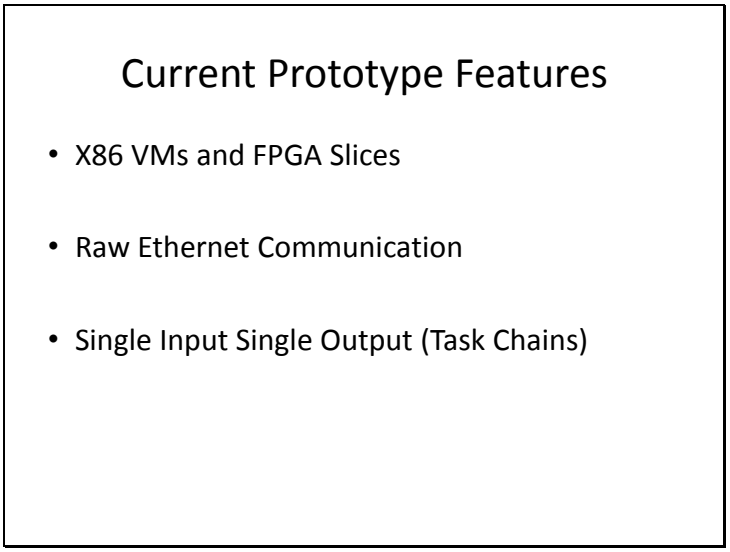
The previous design looks very similar to OpenFlow/SDN. In particular, it consists of a number of computing resources (switches) that forward information based on a globally determined route. The primary differences are:

1) The “action” is a general function based on a piece of aggregate data rather than a packet in OpenFlow
2) Routes are primary determined based on the application, not necessarily the ingress details, although this could be done in OpenFlow as well
3) The computing model treats links such as File I/O as direct transport links. OpenFlow could support inter-process communication by having a software switch and making each process send and receive data from a virtual network port but this adds a layer of extra complexity.

In general, the platform can be thought of as a generalization of OpenFlow.

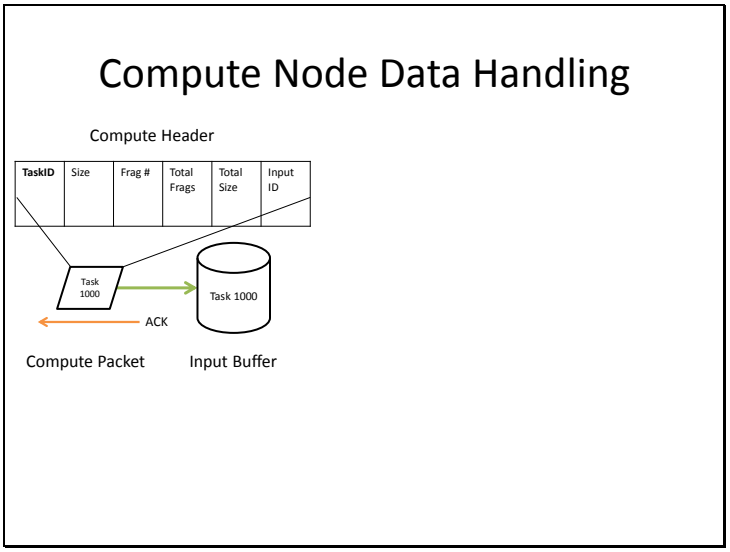
This view is interesting because we can understand how advantages of OpenFlow translate into heterogeneous computing. In particular, OpenFlow switches are very simple since the control is delegated to a controller. This has benefits for systems like FPGAs where control overhead can be expensive in terms of chip area. In addition, the global view allows complex mapping algorithms to assign tasks in a simplified manner rather than having each node attempt to find its best next hop independently.

Slide 14



The prototype designed for this course supports a small subset of the desired features. In particular, the focus was on connecting X86 VMs and FPGAs. Thus, only raw Ethernet is supported as a transport and control layer. Further, only single input single output task graphs can be created. In other words, only chains of computations are allowed.

Slide 15



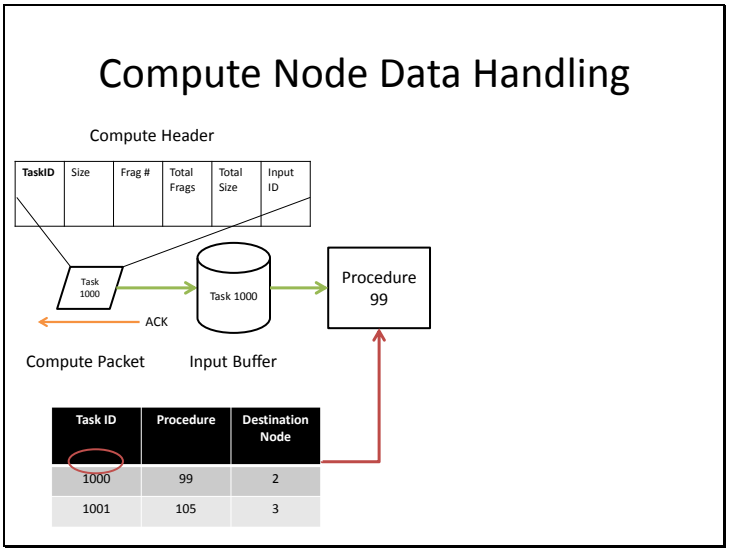
Datapath

Once a route has been established, compute data is sent through the path. Every compute frame has a standard header with a task ID as well as fields describing the fragmentation of a piece of data. Since a computation is designed to operate on a package of data such as a video frame or audio file, the system was designed to support fragmenting and collecting Ethernet frames into packages. The fields such as fragment ID and total fragments are used to piece together out of order Ethernet frames and store them into an input buffer. The task ID defines the forwarding path of the data for this node and will be described in the next page.

Flow control in the system is maintained with a simple handshake. Every Ethernet frame in the system is acknowledged by the receiver. This allows a node to halt incoming traffic by not acknowledging new packets until its buffer has enough space to continue.

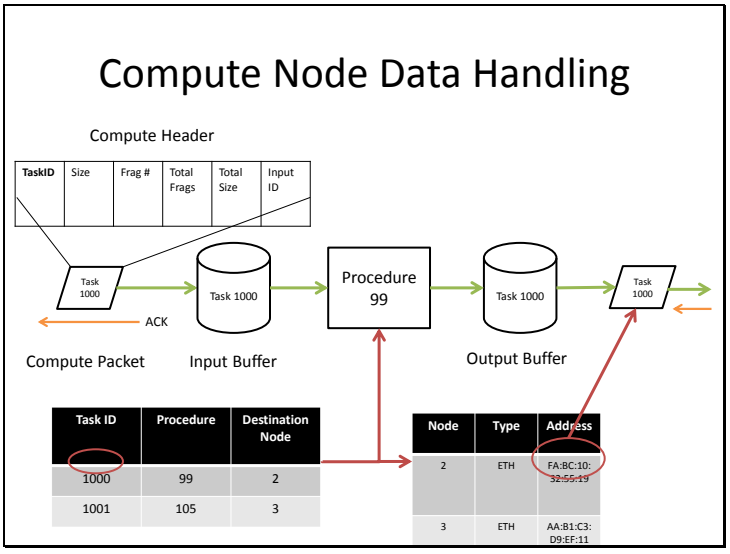
The final field “Input ID” is used to differentiate between data coming on different input paths to the node. For instance a node might wish to add two streams together and so must keep track of which virtual port data arrives at. In the prototype this field is not yet used.

Slide 16



When a compute package is pieced together, its task ID is consulted in a forwarding table that determines the procedure (action) and destination node(s). The hardware performs the procedure defined by the table on the package to produce an output. For now, procedures refer to pre-defined functions in a compute node. Note, a procedure could mean fetch some other data from a table based on the input buffer in the case of content retrieval.

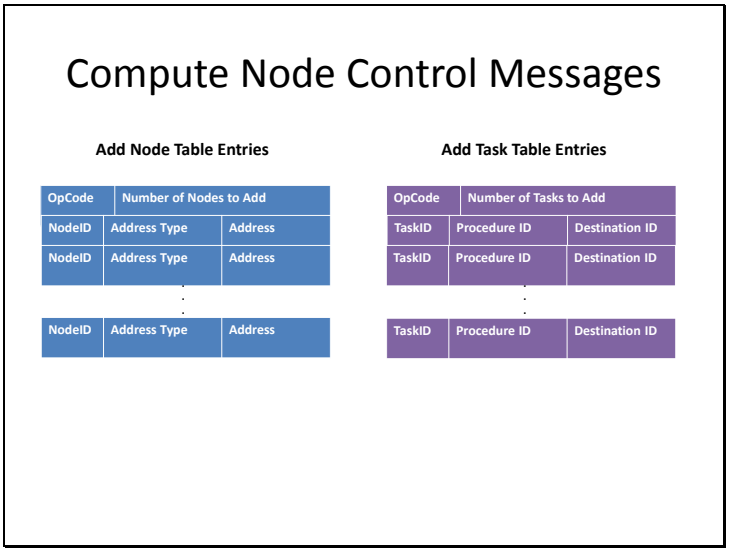
Slide 17



Once the data has been processed, it is passed to an output buffer where the package is broken down into Ethernet frames for the next hop. In this step, a second table is used to translate the destination node to an Ethernet address for transmission.

The idea of having a node ID to address translation was to isolate the resource graph from its physical implementation. In general, one could reach the destination node via different transports so the address could be a 32-bit IPv4, 128-bit Ipv6, 48-bit MAC address or an arbitrary length file reference.

Slide 18

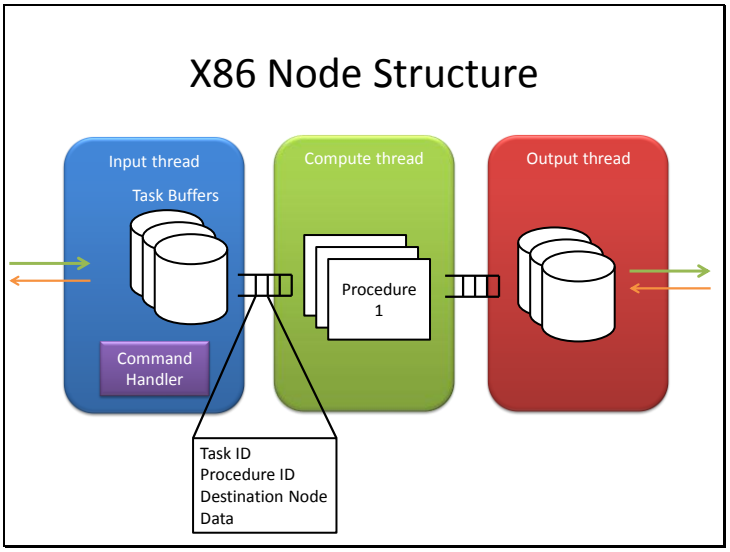


Control Plane

A supervisor can currently control a compute node in two ways. First, it can send an Ethernet frame to add entries to the Node->Address table. Second, it can add entries to the Task table. The control, data and acknowledgement frames are differentiated by different Ethernet types at the moment.

Some missing functionality here is the ability to install procedures and probe counters. In the future, the ability to pass custom procedures that a node would perform is required to support generic computations. For an X86/GPU, the mechanism could be passing a shared object that is linked in with the runtime and called. For an FPGA, the procedure could be a custom bitstream that reconfigures a partition or instantiates a new one with the desired functionality. Statistics counters for current load, link quality, energy consumption and performance on particular tasks are also necessary for a supervisor to efficiently map tasks onto the resources.

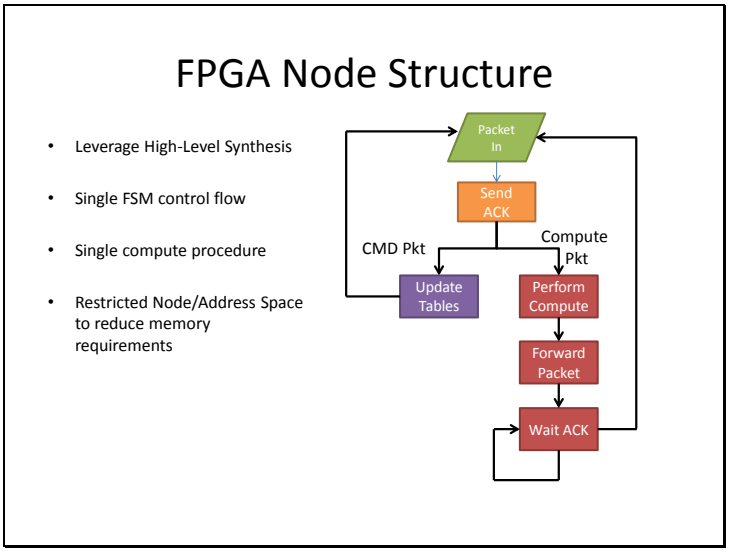
Slide 19



The implementation of the X86 nodes follows the previous discussion closely and was implemented in C++ using pthreads and raw sockets. Three pthreads are used to allow the input, compute and output to operate concurrently. Thus, data can be pipelined through the application efficiently. The input thread allocates memory and reconstructs compute packages from Ethernet frames. It also handles updating the Node and Task tables. Once a compute packet is packaged, its details are passed on to the compute thread. Note that a pointer to the data package is passed and the data is not redundantly copied. The compute thread performs the operation specified by the procedure ID and passed the updated package onto the output thread. Note that currently, only the identity function is supported in the compute thread, although the harness is in place to easily add different transformations. Finally the output thread looks up the destination address via the Node table, shreds the package and sends out Ethernet frames.

Shared memory FIFO queues are used to pass data between threads in a safe manner. With this scheme, there is not queue arbitration and the data is processed in order of arrival to the queue. Future schemes could investigate adding priority to certain tasks and using multiple queues.

Slide 20



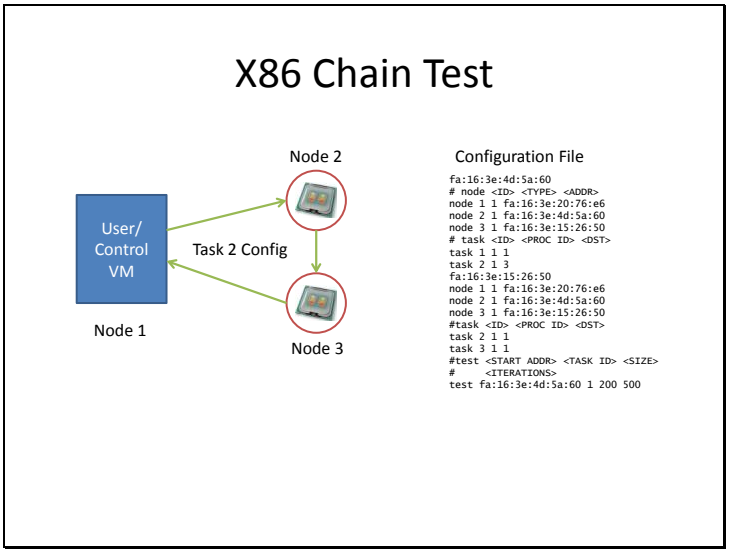
The FPGA hardware was designed using High-level synthesis. This tool converts C code to hardware making development much simpler. A reason to leverage HLS for FPGA nodes is the fact that compute procedures are very easily defined in HLS and a non-hardware designer can create hardware kernels quickly. This is important if many tasks are to be supported on the FPGA hardware.

The current design has some limitations in that the input, compute and output are handled by a single sequential machine. Thus, the hardware handles one Ethernet frame completely before reading a new one. In addition, the Node and Task tables are reduced in size so they consume fewer memory resources.

Many optimizations could be done to this design. In particular, the input and output could be re-written to operate concurrently to the compute. Also, hardware pipelining and refactoring could be used to get more parallelism.

The current node is designed to make use of the virtual FPGA partial regions available in the SAVI testbed and thus has relatively few resources to store a compute package. Experiments could be done to use larger FPGA partitions or different FPGA platforms.

Slide 22

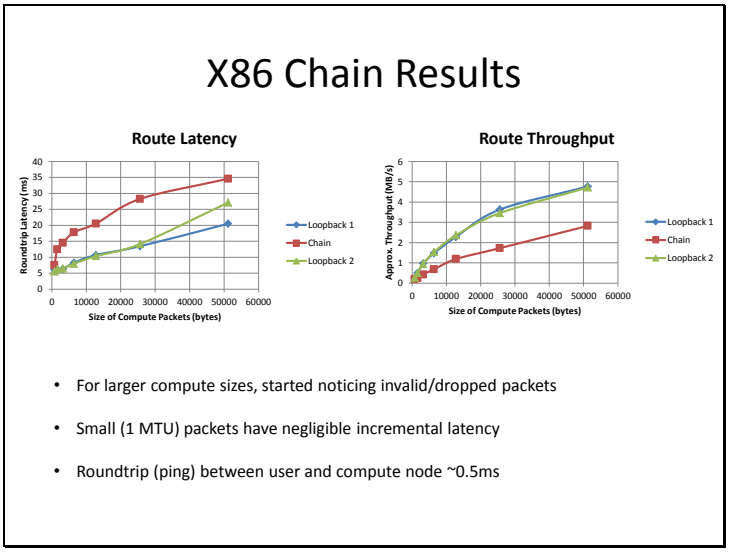


To test the system, a basic test platform was put together chaining two X86 VMs together. The right side of the slide should the configuration file for the test which the controller used to send command frames to each node. Three tasks are defined:

1) Node 1 -> Node 2 -> Node 1 (Loopback 1)
2) Node 1 -> Node 2 -> Node 3 -> Node 1 (Chain)
3) Node 1 -> Node 3 -> Node 1 (Loopback 2)

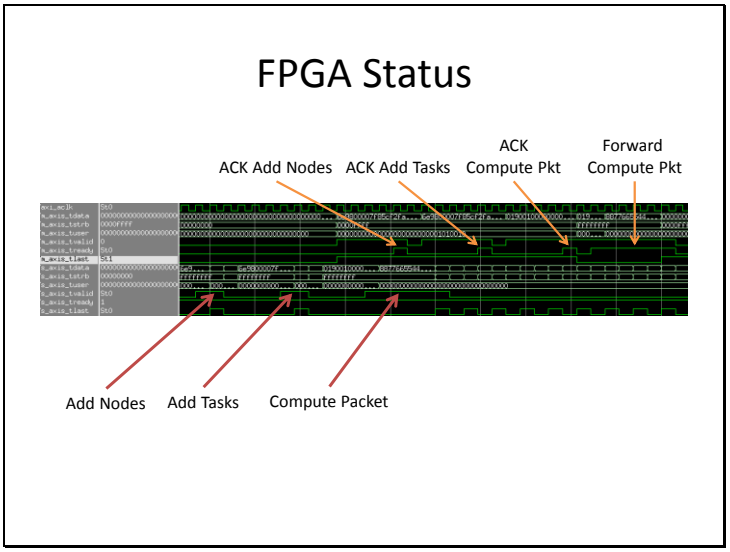
The goal here was to examine the overhead involved in the system infrastructure. Each route was tested for an increasing compute package size over 500 iterations measuring roundtrip latency and throughput.

Slide 23



The results are largely as expected. For the most part, the latency when traversing two compute nodes (Chain) is twice the latency of traversing a single node. Larger packets are streamed across the system leading to an increase in throughput. For instance, it's faster to copy a single large buffer in memory than many small ones. After a package size of about 50 KB, dropped packets were noticed and so results for higher transmission speeds were not reliable. The raw Ethernet protocol provides basic flow control but is not robust to packet loss so the system can stall.

Slide 24



Unfortunately the FPGA partition was not working in hardware. It would have been very instructive to construct a similar latency/throughput graph for FPGA nodes to see where the advantages and disadvantages of the different architectures lie.

The last status on the FPGA accelerator is (perceived) functionality in hardware simulation and the ability to received ACKs on SAVI. However, the translation table is mangled and so the results are not forwarded correctly to the receiver. The slide shows the functionality in simulation. We can see that the acknowledgement paths are operating as nodes and tasks are added. In addition the compute frame is forwarded as expected.

For basic forwarding, there are about 10 cycles between the compute data being received and being sent. Operating at 160MHz, that is 6.25ns per cycle and so a latency of about 62.5ns. Note that this speed is only within the custom logic and the packet still needs to traverse some static hardware and the Ethernet MAC. However, this result is very promising and the FPGA should be able to obtain much lower latency than the X86 VM.

Slide 25

Summary

- Stream Processing Infrastructure over Virtual Heterogeneous Resources
- Initial Prototype of X86 Compute Node
- Partial Prototype of FPGA Compute Node

In summary, the work in this project involved developing a method of mapping streaming programs onto heterogeneous hardware. The proposed architecture turned to be very similar to OpenFlow and it would be very interesting to develop extensions to OpenFlow to support more general stream computing.

The developed system has a functioning X86 compute nodes that can be chained together in an arbitrary topologies using a simple Ethernet command interface. An FPGA compute node has been developed but is not yet functional in hardware.