C-CORE: Using Communication Cores for High Performance Network Services

Sanjay Kumar, Ada Gavrilovska, Karsten Schwan, Srikanth Sundaragopalan
Georgia Institute of Technology
Center for Experimental Research in Computer Systems
Atlanta, Georgia 30332, USA
{ksanjay, ada, schwan, srikanth}@cc.gatech.edu

Abstract

Recent hardware advances are creating multi-core systems with heterogeneous functionality. This paper explores how applications and middleware can utilize systems comprised of processors specialized for communication vs. computational tasks. The C-CORE execution environment enables applications, through middleware and underlying system functionality, to utilize both the computational capabilities of general purpose CPUs and the high performance communication hardware provided by specialized communication processors. Such future heterogeneous multi-core hardware is emulated by attaching a representative network processor – Intel's IXP2400 processor – to a general purpose CPU via a dedicated interconnect. For this platform, C-CORE provides abstractions to represent an application's communication actions, to efficiently couple such actions with application-level computations, and to dynamically create and configure the platform-resident 'chains' of computational and communication actions used by applications. C-CORE's functionality is evaluated with representative, communication-intensive applications. Measurements on our experimental platform establish the performance advantages afforded to applications by C-CORE.

1. Introduction

Large-scale distributed applications increasingly use middleware to efficiently utilize underlying systems' computational and communication services. Middleware, in turn, dynamically deploys processing along the data paths of application-level overlays [16]. Processing ranges from simple data routing and forwarding, to the boolean functions carried out in distributed publish/subscribe [21] infrastructures, to application-specific actions that manipulate, transform, aggregate, and/or distribute information on sets of source-to-sink paths. For instance, in commercial applications like the operational information systems used by airlines [13], middleware services execute simple business rules to transform and route business events between the company's central processing site and remote sites like airport terminals or baggage agents. In distributed scientific collaborations, scientists rely on middleware services to monitor and steer remote experiments, accessing the subsets of experiments' outputs relevant to their current interests, at levels of detail appropriate for their local platforms (e.g., PCs vs. high end workstations) or their communication resources (e.g., available network bandwidth). An important attribute of all such applications is that they exhibit substantial dynamics in terms of data sources and clients, platform resources or end-user interests; this means that overlays and overlay services must be constructed, configured, and tuned at runtime.

Our research goal is to improve systems' abilities to deliver high performance communication services to application-level overlay services. We leverage current hardware trends [4, 6], which indicate that future processors will be multi-core systems comprised of many processing cores (processors) on the same chip. Moreover, there will be both homogeneous (like SMPs but more tightly coupled) and heterogeneous multi-core systems, the latter consisting of different cores optimized for different purposes. One example is IBM's cell processor [4], which has cores specialized for gaming applications. The example considered in this paper is a multi-core platform with computational and communication cores, emulated by using a general purpose CPU with an attached network processor (NP). In particular, we are using an IXP2400 NP attached to a general Intel P4 Linux host through a dedicated PCI bus.

Our approach to improving the performance of middleware-based applications is to enable them to 'best' use the heterogeneous, computational vs. communication-centric processors present in underlying hardware platforms. Toward this end, the C-CORE execution environment provides a set of abstractions (1) to represent an application's overlay services, (2) to enable application-level computations to be executed on different cores, and (3)

to dynamically create and configure the platform-resident 'chains' of services computational and communication actions.

While the basic C-CORE model can be applied to any streaming data application, in this paper, C-CORE is used with applications using the publish/subscribe messaging model [15, 21]. Supporting this model is particularly challenging because of the potential presence of a large number of subscribers to the same information, each of which may require data to be customized before receiving it. Customization is expressed with data filters provided by subscribers that must be executed on messages. A specific example is a set of filters that extract different information from the flight records such as those that extract passenger vs. baggage vs. meal preference information. The role of C-CORE, in this context, is to permit middleware to construct efficient 'processing chains' for application-level messages, and to appropriately and dynamically map such chains across the computational and communication cores of underlying machine platforms. More generally, the distributed brokers [15, 21] or overlay networks constructed by middlewares are mapped to sets of C-CORE execution environments. Middleware functions realized with C-CORE and deployed across distributed nodes execute functionality that includes data distribution, manipulation, and transformation, between data sources and sinks.

The technical contributions of this paper are (1) the C-CORE execution environment supporting the processing of application-level messages across multiple, heterogeneous execution engines on single machines and ultimately, across distributed systems, (2) measurements on a prototype heterogeneous multi-core platform that demonstrate the performance advantages derived from using C-CORE to implement the communication functionality of distributed information flow applications, and (3) experimental results that illuminate what application-level functions are suitable to run on different cores of future multi-core machines. The latter goes beyond standard publish/subscribe message filtering and forwarding functionality, by implementing some of the database-like query operations present in the information flow graphs considered in distributed information flow applications [11].

2. The C-CORE Hardware Platform

Our goal is to create a runtime environment for efficiently carrying out the functions executed by broker (overlay) nodes in content distribution systems like publish/subscribe or distributed information flows. The novel hardware platform being considered is comprised of both general purpose and specialized communication cores, termed g- and c-cores, respectively. G- and c-cores are tightly coupled, able to efficiently access shared memory, and each c-core can interact with multiple physical network interfaces. Three assumptions underlie this research. First, it assumes that g- and c-cores are tightly coupled, more so than current combined host-NP architectures, implying, for example, the ability to share certain cache resources. Second, it assumes that c-cores will have some of the characteristics of current network processor architectures, specialized to efficiently execute communication stacks. Third, it is not concerned with other specialized cores in future machines, such as those focused on graphics or storage tasks.

2.1 Communication Core

c-core that is similar to those of current NPs, with many internal processing units, each independently programmable and with sufficient resources to support a single communication stream at link speed. In Intel's IXP NPs, for instance, the c-core has hardware processing units termed microengines that can operate in parallel, have their own registers and small amounts of program memory, and have shared access to hierarchically arranged memories of different speeds. These engines are programmed such that one or more of them can be allocated for control plane operations, and others can be allocated for data-plane operations. In addition, engine actions can be chained to form processing pipelines that implement more complex messaging operations. Engines have direct access to high speed network interfaces, with additional hardware present (in some IXP processors) for specialized processing tasks like encryption.

3. Using C-Cores in a Middleware System

This section presents an architecture that enhances the capabilities of traditional publish/subscribe middleware, making it 'c-core aware', by enabling it to dynamically create, deploy, and configure c-core processing.

Programming c-cores is not a task for arbitrary application developers. Instead, as with mathematical libraries supported by FPGA coprocessors in modern Cray machines [5], c-cores will typically be used with libraries that implement standard communication functionality. The contribution of our work is to go beyond such library-based c-core usage to also enable them to execute application-specific functionality, by using middleware as the 'mediator' between applications and communication processors. The goal is to permit c-cores to perform meaningful application-specific actions, thereby permitting applications to directly leverage their abilities to run at physical link speeds, *close* to the physical network, tightly linked with standard communication processing actions, and utilize hardware optimized for communication processing [14, 20] (e.g., multiple hardware queues, direct low latency access to physical network links, etc.).

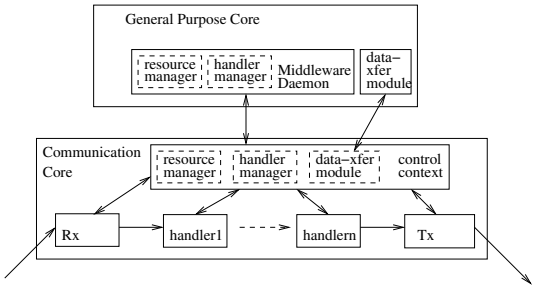

Figure 1. Architecture overview: communication-core aware middleware system.

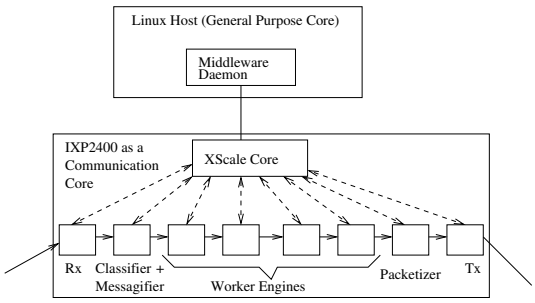

Figure 2. Architecture overview: using IXP2400 in a c-core aware middleware system.

The C-CORE software environment for executing application-level functions on communication cores has two main components, one resident on the generic computational core (g-core), the other resident on the communication core (c-core). These two components permit applications, via middleware, to establish pipeline-structured sets of communication actions that span g- and c-cores. On the g-core, such actions are executed at application-level, using a reservation-based approach to give hosts access to certain communication resources, and use a specialized network driver to provide what appears to be a standard network connection between g- vs. c-core-resident communication actions. The remainder of this section describes the C-CORE environment's main components or modules, the functionality it provides, and its programming abstractions.

3.1. C-CORE Software Architecture

Figure 1 depicts the main components of the C-CORE software architecture. The architecture provides c-core resident *processing contexts* able to run middleware-provided *handlers*. Processing contexts and handlers are managed by a set of components that include:

- a *resource manager* that monitors the c-core resources (buffers, CPU cycles, network bandwidth) used by handlers running on processing contexts and reports to the g-core to reflect current c-core load; handlers are classified into communication- and computation-centric ones;
- a *handler manager* that controls handler deployment, implements handler hot-swapping, and manages handler state information; and
- a *data transfer module* that handles packet transfers between g- and c-cores.

A typical set of actions carried out by these components is the following. When the publish/subscribe middleware used in our experimentation receives a subscription request from a new client, along with a handler implementing the subscription's logic, middleware can choose to install the handler on the g- or c-cores of its publish/subscribe broker node, or, in the more sophisticated publish/subscribe middleware now being developed by our group, middleware can dynamically compile handler functionality into some suitable chain of primitive handlers mapped across both g- and c-cores [1]. In either case, the C-CORE software architecture's role is to support the dynamic creation and management of such handler chains. The sequence of actions taken to install a chain begins with a request that checks the c-core's current work load and resources (e.g., its buffers, CPU cycles, etc.), and if affirmative, creates a handler chain by installing suitable handler representations on g- and c-cores. The aforementioned data transfer module handles packet transfers between the two cores. Another option is to create a new sub-channel for an already established communication channel, where a subscriber simply provides a new handler, which is then used to install an additional handler chain across suitable g- and c-cores associated with the same communication links.

Since handlers process application-level messages, the C-CORE execution environment implements basic protocol processing functionality, including message fragmentation and reassembly and in our current implementation, a variant of reliable UDP running on one of the IXP microengines does message fragmentation and reassembly. In addition, C-CORE maintains information about the type and structure of application-level messages, using an efficient binary message format developed in our previous research [3]. In ongoing work, we are also considering binary XML message formats. In any case, C-CORE provides sufficient functionality to essentially make c-cores 'message-aware', by maintaining message format information passed by the data source to the C-CORE during channel creation and then using such information for message assembly and for allowing handlers to interpret message contents. We note that C-CORE only assembles those messages to which application-level handlers must be applied, thereby avoiding such overheads for other data flowing through a c-core.

Dealing with heterogeneity. Since g- and c-cores are optimized for computational vs. communication tasks, we similarly separate the handlers executed by publish/subscribe middleware into computation- vs. communication-centric code fragments. Our current work manually identifies the nature of handlers, relying on developers to determine possible handler execution sites. [7] describes automatic methods for determining whether certain handlers are suitable for c-core execution, at minimum involving assessments of handler resources requirements and corresponding resource availabilities on c-cores. Our future work will develop compiler-based techniques to dynamically generate handlers and determine their appropriate execution sites.

Handler and data formats. Our previous work [8] has developed the basic functionality needed to represent application-provided code on c-cores, described as *stream handlers*, and to efficiently describe the structure of application-level messages via *message formats*. This paper uses stream handlers and message formats from [8], but extends that functionality by (1) providing the ability to hot-swap stream handlers, (2) implementing an explicit representation for the chains of handlers applied to the end-to-end stream data path across g- and c-cores, termed *service paths*, and (3) realizing data structures that explicitly represent the different message formats used by handlers. The idea of (3) is to permit a single handler to process messages with differing formats.

C-CORE relies on information about message formats embedded in application-level messages. These formats specify data size and layout, offsets, and sizes of the application-level data fields that comprise the data unit. The actions applied to a message are determined by a combination of data format and network-level information, which is maintained in a format cache, used by handlers to access specific data fields. Stream handlers are used to implement C-CORE 's operators applied to information streams. By dynamically deploying operators across g- and c-cores and by chaining and configuring them, we construct linked service paths, described by execution contexts that execute operators and the formats of stream data accepted by operators.

Dynamic resource management and reconfiguration. The resource manager on the c-core monitors current resource usage and availability. On g-cores, middleware discovers all c-cores and their states through querying them for their current resource availabilities and usage. We expect this functionality to balance and dynamically reassign workload across different c-cores. Dynamic assignment can involve runtime code generation and specialization. The C-CORE execution environment supports this by enabling runtime handler configuration and hot-swapping. This paper uses these capabilities to better match the behavior and performance of c-core handlers to current application needs.

An effective way to improve the performance of resource-limited c-cores is to dynamically specialize message handlers. Specifically, consider a generic deployed handler. Such a handler must read information about data formats and/or the current parameters to be used when interpreting application-level data from memory. However, if the handler is going to operate on some particular stream with fixed parameters, the handler manager can rewrite the handlers with the stream format and parameter information hardcoded in its implementation. Alternatively, format and parameter information can be cached in processor registers. Experiments show that such specialization has substantial benefits for computationally intensive stream handlers.

4. Implementation

Our prototype implementation of the C-CORE architecture uses a Linux host to represent a g-cores, and IXP2400 network processors [10] to represent a c-core, both interconnected via a dedicated PCI interface. The 8 microengines available on the IXP2400 provide the C-CORE architectures processing contexts and run either dedicated components of the execution environment like message fragmentation and reassembly or provide processing contexts available for executing middleware-provided stream handlers. The XScale core on the IXP runs control and initialization operations, facilitates data transfers across the PCI interface, and performs c-core reconfiguration (e.g., deployment of new handlers on the c-core). As shown in Figure 2, all 8 microengines are assigned to different tasks, able to work in parallel on different message streams and/or in a pipelined fashion for a single stream. In our current implementation, four microengines remain available to execute handlers or chains of handlers on application-level messages. We expect future c-cores to have additional execution engines available for application-level processing tasks.

The remainder of this section describes the elements of our implementation needed to help explain and motivate the experimental results presented in Section 5, including (1) how we describe the structure of application-level messages and how handlers access messages and use such structure information and (2) how handlers are chained to enable more complex operations on message contents. (3) We also describe the implementation of handler hot-swapping.

Understanding application-level messages. In order to enable the association of handlers with application-level messages, we rely on (1) IP header information and sequence numbers to assemble the message, (2) the PBIO [3] representation of the binary message data format to understand message layout in memory, and (3) unique stream identifiers. The stream identifier represents a combination of source-destination address pair and port number and the PBIO format id. This triple is used to classify application-

level messages. Based on classification outcome, messages may be transferred to the g-core, or they may be queued to a queue corresponding to the format id for additional processing on other microengines. There are different queues for different format ids, and a particular handler picks up messages from the queue corresponding to the format identifier on which it is operating.

Handlers and their execution environment. Handlers are specified in microcode object format (.uof), which has the capability to represent multiple processing contexts and also has information about which processing context should be run on which microengine. The loader reads this information from the handler header and copies the specific contexts to their respective microengines' instruction memories. The current implementation of C-CORE assumes that handlers are provided by the upper middleware layers or sent by subscribers (clients), as precompiled binary executables. We are now extending C-CORE with a dynamic code generation mechanism to recognize handlers provided by clients in some higher level language (e.g., Microengine C) and dynamically call the appropriate compiler to generate executable code.

Pipelined implementation of handler chains. Representing more complex application-level processing actions as sets of chained handlers enables us to split the execution of time-consuming processing across multiple microengines, to gain pipeline parallelism, and to prevent any one execution context from becoming a bottleneck. The C-CORE pipeline implementation uses non-copying message queueing between different pipeline stages and implements an efficient handoff protocol.

Processing deployment on IXP. In order to deploy a stream handler, C-CORE relies on runtime resource management and on information provided with the handler, to determine correct and suitable handler deployment. If the stream handler is to be deployed on a specific context on the c-core, the handler manager (the XScale in this case) initiates the reconfiguration, maps the handler to the appropriate microengine(s), copies the handler code to its instruction memory, initializes the corresponding state and data paths, and kick-starts the 'new' microengine (pipeline).

Dynamic reconfiguration through hot-swapping. In order to best utilize resources and match current application needs and platform resources, handlers need to be deployed and configured dynamically (hot-swapping of handlers). This can be initiated in response to changes in system resources or in end user interests. Simple reconfigurations involve changes to stream handler parameters, such as changing the selection criteria in a select query handler. More complex configurations involve hot-swapping stream handlers, including to address limitations in the amount of instruction memory in the microengines. Hot-swapping of handler code is also required for handler specialization

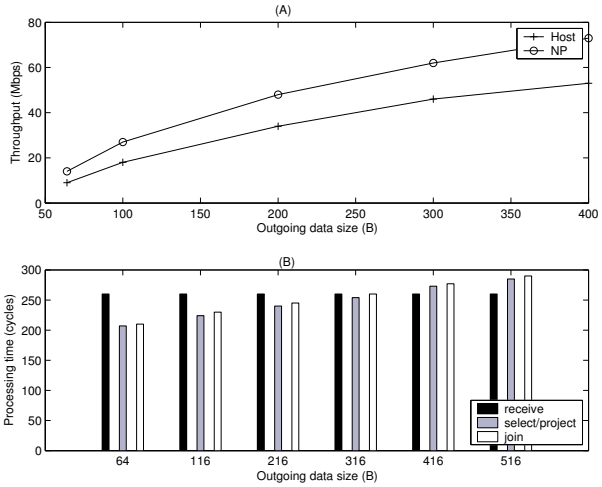

Figure 3. A) Query throughput on g-core vs. c-core, and B) Processing time of various pipeline stages

where a handler is redesigned to specialize it for its current execution environment and redeployed in place of its older version. Automatic handler specialization, based on runtime monitoring as in other specialization systems has not yet been implemented.

The current implementation of hot-swapping keeps one of the microengines in idle state, while others are used to run handlers. During hot-swapping, the new handler is loaded onto the idle microengine and then the control is switched from the old microengine to the new one. The idle microengine is started while the old microengine is stopped and becomes the idle microengine to be used for next hot-swap. This implies that the actual downtime for handler processing is equivalent to the costs of stopping one microengine and starting the other one. Measurements show that this can be done in about 30 microseconds as compared to around 400 microseconds when the same microengine is stopped, new handler code is loaded and then restarted. The drawback of this method is that one of the IXP2400 microengines must be kept idle for hot-swapping.

5. Evaluation

We next present an experimental analysis of the prototype C-CORE implementation on Linux hosts and their attached IXP2400s. The experimental testbed is comprised of 8 Dell Poweredge 2650 Machines (4 Xeon 2.8 GHz each), each having one IXP2400 (Radisys' ENP2611 board) attached to it via a PCI interface. IXPs are interconnected via a Gigabit LAN. Results demonstrate the importance and feasibility of enabling the execution of application-level data manipulations on c-cores. They also analyze the processing costs and specific benefits derived from the c-core execution of select handlers representing commonly used

publish/subscribe services.

Feasibility of handler execution. The first set of experiments evaluates the c-core's ability to execute middleware-provided handlers. Our evaluation uses an implementation of 'continual database queries' [11], comparing its performance with a corresponding g-core based implementation. Queries implemented by operators are applied to streaming data in order to create customized/personalized representations of that stream for different clients. The operators evaluated are the database operators (i.e., select, project, join) used in publish/subscribe infrastructures like IBM's Gryphon [21] product. The following specific test case is used.

Two publishers generate data streams and send them to a single *broker*. In addition, two subscribers submit queries to the *broker*, one query doing select/project on individual streams and the other doing join operation on the two streams. Total three sub-streams are generated, two corresponding to individual streams and the third is join sub-stream. In the c-core based implementation, the *broker*'s g-core receives the query handler from a subscriber and deploys it on the c-core. The query operators are applied to data streams that carry data from the operational information system of an airline (see [13] for more detail on that application). These data streams are directly sent to the c-core via its gigabit links. Similarly, the sub-streams produced by the c-core are directly sent, via its output links, to the subscribers that desire them. On the c-core, operators are executed in pipelined fashion. In comparison, the g-core implementation deploys query operators on the g-core, using a multithreaded approach, where the same message streams are processed with the same select, project, and join operators as those used in the c-core scenario.

A representative query for stream A is as follows: *select passengerList, mealPreference from A where A.source="Atlanta" AND A.destination="Paris" AND A.departTime="20:40 pm"*. Note that query operators can substantially differ in complexity, where complexity not only derives from the number of conditions tested and evaluated, but also from the number of different message fields accessed, the sizes of such fields, and the sizes of the messages created for output sub-streams.

The experiment shown in Figure 3.A compares the execution of a set of these queries on c- vs. g-core. The results shown are the attained throughput for c-core and g-core for different query complexities. We observe that for all data sizes, the c-core is capable of delivering improved throughput levels compared to the host. Note that the reasons for these improvements are complex, involving both the innate differences in hardware structures and capabilities on g- vs. c-cores and the general vs. specialized nature of g- vs. c-core execution environments, in the critical data path involving an entire Linux OS on the g-core vs. minimal runtime support on c-cores. Additional results on the IXP's capabilities to execute selected application-level handlers appear in [7].

Ability to deploy handler chains. We next analyze the ability of constructing and deploying query chains on the IXP c-core.

Figure 3.B presents time in terms of the processing cycles required by different components of the application, and by different query handlers (for different message sizes). We observe that query operations can be executed in approximately the same amount of time as the receive operation for output data streams of the same size. This demonstrates that the c-core is capable of supporting chained queries. Moreover, we can hypothesize from these results that when operators reduce the amounts of data produced vs. received (e.g., a sub-stream contains a subset of the information contained in the arrival stream), similar performance results will be attained. We note that of course, by executing stream processing on the c-core, the g-core is freed to carry out other application-level tasks.

We next determine the limits to which the IXP2400 can sustain query processing by increasing stream throughput. The case measured constitutes a worst case in that it does not reduce the size of output compared to input data (using 600 byte messages). Table 1 shows that the IXP2400 can sustain close to 350 Mbps of throughput. Beyond that, packet drops become significant. We expect to be able to sustain higher levels of throughput for queries that lead to reduced size output streams (approximating link rates). Measurements also show the overheads incurred in query execution. In fact, overheads increase as the input speed increases because of memory and other resource contention issues. Please note that the output throughput is more than the input throughput in some cases. This is because we send two input streams (of 100000 packets each) but output three sub-streams, two corresponding to the individual streams and the third sub-stream as a result of a join query operation on the two streams.

Scalable data distribution services. Multicast protocols are similar to publish/subscribe middleware in nature due to their one-to-many semantics. We have implemented an application-level multicast service using the architecture described in this paper. The service implements subscription channels by building an application-level multicast overlay. Specifically, the C-CORE environment on the g-core of the *broker* host builds a multicast table and passes it to the c-core. This table is then shared with the microengines executing the multicast handler. Messages are assembled, classified into streams, and passed to the multicast handler, which forwards them to every address in the destination table. The implementation of the multicast services is described in greater detail in [18].

I/P Thrput	O/P Thrput	Overhead(usec)	pkts sent	pkts processed	pkts dropped
200 Mbps	200 Mbps	31	200000	255000	0
350 Mbps	350 Mbps	32	200000	230000	20000
495 Mbps	450 Mbps	35	200000	160000	80000
695 Mbps	450 Mbps	50	200000	160000	105000

Table 1. Query throughput and overhead calculation for various input stream speeds on the IXP2400.

I/P Rate	O/P Rate	Time per Pkt	I/P pkts	O/P pkts	dropped pkts
50 Mbps	250 Mbps	1.6 usec	100000	500000	0
100 Mbps	500 Mbps	16.4 usec	100000	500000	0
150 Mbps	750 Mbps	39.8 usec	100000	500000	0
200 Mbps	1000 Mbps	42.8 usec	100000	469246	6150

Table 2. Multicast throughput and overhead calculation for 5 destinations on the IXP2400.

The next set of measurements evaluates the limits to which the IXP2400 can sustain multicast throughput (for 5 destinations) as input speed is increased. Table 2 shows that the IXP2400 can sustain approximately 1 Gbps of output rate (5 destinations) without dropping packets, which is the IXP's line rate. Please note that the dropped packets are the input packets. The conclusion is that this c-core can sustain almost 1 Gbps speed for application-level multicasting. Further experiments are needed to evaluate multicast combined with customization operations for this c-core. Additional results comparing the ability of the IXP c-core to enable data distribution services to the general purpose host, demonstrate that the c-core implementation can sustain throughput levels as the number of destination increases, while with the g-core implementation performance rapidly degrades.

Effects of handler specialization. Our final measurements demonstrate the importance of dynamic handler specialization. Table 3 shows the specialized handler's (compared to Table 1) performance when the stream format and query parameters are hardcoded into its implementation. Clearly, the specialized handler can sustain much higher throughput (closer to 700 Mbps). This is because we are avoiding reading format and parameters information from memory for every message, thereby also reducing potential memory bus contention for other memory accesses. Having demonstrated the importance of this functionality, we are now building a mechanism to cache stream format and handler parameters, avoiding memory reads and taking advantage of the improved performance of specialized message handlers.

6. Related Work

The work presented in this paper builds on our previous research to create integrated platforms of hosts and attached network processors (NPs), so as to enable the execution of application-specific services onto the programmable NP and closer to the network [7]. The programmability of network processors has been widely exploited in both industry and academia, for delivering more flexible network- and application-level services [9, 19]. Our current work assumes that these integrated host-NP platforms exemplify future heterogeneous, multi-core systems.

The utility of executing compositions of various protocol- vs. application-level actions in different processing context is already widely acknowledged. Examples include splitting the TCP/IP stack across general purpose processors and dedicated network devices, such as network processors, FPGA-based line cards, or dedicated processors in SMP systems [2, 17], or splitting the application stack as with content-based load balancing for an http server or for efficient implementations of media services. Similarly, in modern interconnection technologies, the network interfaces represent separate processing contexts with capabilities for protocol off-load, direct data placement, and OS-bypass [12, 20]. In addition to its focus on multi-core platforms, our work differs from these efforts by enabling and evaluating the joint execution of networking and application-level operations on communications hardware, thereby delivering additional benefits to applications.

Finally, the services targeted by the architecture described in this paper are those of publish/subscribe middleware systems, such as those described in [15, 21]. However, the C-CORE architecture is sufficiently general to apply to a wide range of distributed or Grid applications that rely on the ability of underlying middleware to deliver and distribute 'useful' data to select nodes in the application overlay, and to transform the application data stream 'in-transit'.

I/P Thrput	O/P Thrput	Overhead(usec)	pkts sent	pkts processed	pkts dropped
330 Mbps	445 Mbps	15	200000	279000	0
495 Mbps	705 Mbps	11	200000	275000	2000
695 Mbps	705 Mbps	12	200000	240000	6000
830 Mbps	701 Mbps	12	200000	203000	10000

Table 3. Query throughput and overhead calculation for a specialized handler.

7. Conclusions and Future Work

The goal of the C-CORE software architecture is to efficiently exploit the communication cores of future heterogeneous multi-core systems. The architecture is shown to offer improved levels of performance for applications written with publish/subscribe middleware compared to solutions that use standard processors and operating systems. Improvements are attained by permitting middleware to run application-specific functions on c-cores, as single client-provided handlers or as complex handlers realized as processing pipelines. Efficient implementations of continual database queries and of application-level multicast are used to evaluate the architecture.

The C-CORE software architecture is not yet complete. Still being implemented are (1) the mechanism to dynamically characterize a handler as computation vs. communication centric and (2) the mechanism to safely execute client-provided handlers on c-cores. Further, our current implementation is integrated with publish/subscribe middleware, but it is now being generalized to work with a wider variety of information flow middleware and applications.

There is significant security risk in permitting client-provided handlers to run in the communication core. While general purpose hosts offer efficient hardware techniques to address this issue, the IXP2400 used in this work does not have the hardware needed for full isolation across different processing contexts. The implementation of isolation for handlers and for the state they use is an important topic of future research.

References

[1] S. Agarwala, G. Eisenhauer, and K. Schwan. Morphable Messaging: Efficient Support for Evolution in Distributed Applications. In *Proc. of the CLADE Workshop*, 2004.

[2] F. Braun, J. Lockwood, and M. Waldvogel. Protocol wrappers for layered network packet processing in reconfigurable networks. *IEEE Micro*, 22(1), 2002.

[3] F. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient Wire Formats for High Performance Computing. In *Proc. of Supercomputing 2000*, Dallas, TX, Nov. 2000.

[4] Cell Processor Architecture Explained. http://www.blachford.info/computer/Cells/Cell0.html.

[5] *Cray XD1 Overview*. http://www.cray.com/products/xd1/.

[6] *Intel Dual-Core Processor*. www.intel.com/technology/computing/dual-core/.

[7] A. Gavrilovska. *SPLITS Stream Handlers: Deploying Application-level Services to Attached Network Processors*. PhD thesis, Georgia Institute of Technology, 2004.

[8] A. Gavrilovska, S. Kumar, S. Sundagaropalan, and K. Schwan. Platform Overlays: Enabling In-Network Stream Processing in Large-scale Distributed Applications. In *Proc. of NOSSDAV 2005*, WA, 2005.

[9] J. Guo, J. Yao, and L. Bhuyan. An Efficient Packet Scheduling Algorithm in Network Processors. In *Infocom*, 2005.

[10] Intel Corporation. *Intel Network Processor Family*. developer.intel.com/design/network/producs/npfamily/.

[11] V. Kumar, B. F. Cooper, Z. Cai, G. Eisenhauer, and K. Schwan. Resource-Aware Distributed Stream Management using Dynamic Overlays. *Proc. of ICDCS-25*, 2005.

[12] P. Mehra. Apsara: The Quest for the Perfect Server for Network Computing Applications. In *Proc. of NCA-3, keynote address*, Cambridge, MA, 2003.

[13] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin. Operational Information Systems - An Example from the Airline Industry. In *First Workshop on Industrial Experiences with Systems Software (WIESS)*, Oct. 2000.

[14] F. Petrini, J. Fernandez, A. Moody, E. Frachtenberg, and D. Panda. NIC-based Reduction Algorithms for Large-scale Clusters. *International Journal of High Performance Computing and Networking (IJHPCN)*, 2005.

[15] P. Pietzuch and S. Bhola. Congestion Control in a Reliable, Scalable Message-Oriented Middleware. In *Proc. of Middleware 2003*, Rio de Janeiro, Brazil, 2003.

[16] B. Raman and R. Katz. An Architecture for Highly Available Wide-Area Service Composition. *Computer Communications Journal*, May 2003.

[17] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. In *Hot Interconnects*, 2003.

[18] S. Sundaragopalan, A. Gavrilovska, S. Kumar, and K. Schwan. An Approach Towards Enabling Intelligent Networking Services for Distributed Multimedia Applications. In *Proc. of IMMCN'05*, 2005.

[19] K. Yocum and J. Chase. Payload Caching: High-Speed Data Forwarding for Network Intermediaries. In *Proc. of USENIX Technical Conference*, Boston, MA, June 2001.

[20] X. Zhang, L. N. Bhuyan, and W. chun Feng. Anatomy of UDP and M-VIA for Cluster Communications. *Journal on Parallel and Distributed Computing*, 2005.

[21] Y. Zhao and R. Storm. Exploiting Event Stream Interpretation in Publish-Subscribe Systems. In *Proc. of ACM Symposium on Principles of Distributed Computing*, Aug. 2001.