Lagniappe: Multi-$\star$ Programming Made Simple

Taylor L. Riché, Greg Lavender, and Harrick M. Vin
Department of Computer Sciences
The University of Texas at Austin
1 Univerity Station, C0500, Austin, TX 78712
`riche,lavender,vin@cs.utexas.edu`

Abstract: The emergence of multi-processor, multi-threaded architectures (referred to as *multi-$\star$* architectures) facilitates the design of high-throughput request processing systems (e.g., multi-service routers for GENI [19], intrusion detection systems [22], graphics and gaming systems [1, 11, 17], as well as high-throughput web servers and transaction processing systems [2, 3, 16, 21]). Because of the challenges in programming such architectures realizing this promise has proved to be difficult. In this paper, we describe the design of *Lagniappe*, a programming environment that simplifies the design of portable, high-throughput applications on multi-$\star$ architectures. Lagniappe uses a *hybrid* programming model: it combines a *procedural* specification (e.g., in C++) of the basic operators for processing requests with a *declarative* specification—expressed using a *model-driven development framework*—of the various features of the operators and the target hardware platform. Using the declarative specification, the Lagniappe programming environment automates the mapping of applications onto the multi-$\star$ platform, performs dynamic allocation of resources to operators, and ensures efficient and coherent accesses to persistent, shared state.

1 Introduction

Moore's law and the accompanying improvements in fabrication technologies (90 nm to 65 nm and beyond [7]) have increased the number of transistors available to processor designers. During the past five years, processor designers have begun utilizing these transistors to add multiple levels of parallelism (in the form of multiple processor cores, each with support for multiple hardware threads) on a single chip. Today, such multi-$\star$ processor architectures are everywhere, starting with special-purpose processors—such as network processors [6, 23], graphics processors [17], and processors for gaming systems [1, 11]—and now moving to general-purpose processors [15, 24]. Further, system designers are combining such processors to design highly parallel platforms.

A common characteristic of multi-$\star$ architectures is that each of their cores are simpler and slower (with respect to clock speeds) than conventional processors. Unlike conventional processors, however, multi-$\star$ architectures provide significant numbers of parallel resources, making them ideal for *request processing* applications. Request processing applications exhibit large amounts of task parallelism because requests often can be processed in parallel. Although much of the following discussion is independent of the type of request processing application, for the remainder of the paper we focus on *packet processing systems (PPS)* on multi-$\star$ platforms.

PPS are designed to process network packets efficiently. A PPS supporting multi-gigabit network links generally processes millions of packets per second. During the past several years, the diversity and complexity of applications supported by PPS have increased dramatically. Examples of such systems include multi-service routers for GENI, Virtual Private Network (VPN) gateways, intrusion detection systems, content-based load distribution, and protocol gateways (for example, an IPv4/v6 gateway).

Although multi-$\star$ architectures are well-suited for PPS, realizing the vision of designing easy-to-program, high-throughput PPS has proven to be difficult because of the difficulties in programming multi-$\star$ platforms. These difficulties arise from three factors:

- There are at least three different ways to map a packet processing application onto a multi-$\star$ system. The *pipeline* approach splits the application into independent stages and maps each stage to a processing element; thus, each packet during its lifetime traverses multiple processing elements. The *parallel* approach lets each element process a packet from start to finish; processing elements available in a multi-$\star$ system process multiple packets in parallel. Finally, the *hybrid* approach replicates some parts of the application while staging others. Choosing the approach that delivers the highest throughput is hard because the choice depends upon application, system, and workload characteristics [18].

- Most packet processing applications of interest are stateful. In particular, these applications maintain persistent state that is accessed and updated by a stream of related packets (often referred to as a *flow*). In a multi-$\star$ system with multiple distributed memory levels and message passing channels, providing efficient and coherent access to shared state is challenging. Further, the non-uniform memory architectures of many multi-$\star$ systems complicates the selection of an appropriate policy (e.g., packet-level vs. flow-level with flow-pinning) for distributing packets across processing elements [18].

- Each application using a multi-$\star$ system generally processes multiple types of packets. In most realistic deployments, however, the workload (both the composition of packet types and volume of traffic) fluctuates significantly over time. Hence, to process different types of packets a PPS must adapt the resource allocations made by the programmer [14].

Today, programmers of packet processing systems built using multi-$\star$ platforms must manually address each of these challenges. To enable programmers to perform these tasks, processor and system designers expose all the low-level architectural details to programmers (e.g., Intel's IXP2800 programming environment [23]). However, attempting to map computation, handle stateful applications, and adapt resource mappings on exposed low-level hardware leads to tedious and error-prone programming. To make matters worse, the resulting software is non-portable.

In this paper, we describe the design of *Lagniappe*[1], a programming environment that simplifies the design of portable, high-throughput packet processing applications on multi-$\star$ systems. Lagniappe uses a *hybrid* programming model: It combines a procedural specification (e.g., in C++) of the basic operators for processing packets with a declarative specification expressed using a *model-driven development framework*. The declarative half specifies various features of the operators and the target hardware platform that allow Lagniappe to automate the three main challenges of multi-$\star$ development: (1) mapping of applications onto the multi-$\star$ platform, (2) ensuring efficient and coherent accesses to persistent shared state, and (3) performing dynamic allocation of resources to operators.

The rest of the paper is organized as follows. In Section 2, we describe the programming model for Lagniappe; in particular, we focus on the declarative part of Lagniappe and describe the meta-models for formally capturing the features of applications and multi-$\star$ systems. We describe our model-driven code generation environment in Section 3. In Section 4, we walk through an example application to show the flexibility, simplicity, and power of Lagniappe. We review related work in Section 5 and summarize our contributions in Section 6.

2 Programming Model

The Lagniappe programming environment must meet three requirements: (1) automate the mapping of applications onto multi-$\star$ platforms, (2) provide efficient and coherent accesses to persistent operator state, and (3) perform dynamic allocation of resources to operators as workload changes. To satisfy these requirements, Lagniappe uses a hybrid programming model that combines a procedural specification of the basic operators for processing packets with a declarative specification of the features of the operators and the target hardware platform.

The model achieves *separation of concerns* [10]—the design of operators (that implement basic packet processing functionality) is decoupled completely from the design of applications (i.e., how the basic operators are composed to construct an application) as well as the issues involved in mapping the application onto multi-$\star$ architectures. This both simplifies the programming task and facilitates modularity and reuse of operators.

Procedural specification of operators Operators are developed in a general-purpose language (e.g., C++) familiar to programmers. The Lagniappe programming model enforces only one requirement for the operator designs: The code for the operators must be *thread-safe*; access to persistent state maintained by the operator must be protected using locks and condition variables. This requirement ensures correct operation of the operator regardless of the number of instances of the operators that may run concurrently on a multi-$\star$ platform. Because operator designers are completely unaware of the configuration of the multi-$\star$ platform, the operators are portable.

Declarative specification of applications and systems Lagniappe uses a *model-driven development (MDD)* framework to formalize the specification of operators, applications, and multi-$\star$ platforms. Model-driven development is known to have advantages in terms of productivity and portability over traditional low-level development techniques [12]. In an MDD framework, one creates *meta-models* to create a modeling language; a designer creates *instances* of these models to describe applications and multi-$\star$ systems. These instances then can be used to generate platform-specific code.

For Lagniappe, we define two meta-models: (1) an *application meta-model* and (2) a *system meta-model*. In what follows, we first describe these meta-models and then outline, in Section 3, how the Lagniappe programming environment uses these models to achieve the three earlier stated goals.

2.1 Application Meta-Model

The application meta-model in Lagniappe consists of two parts: (1) specification of persistent state maintained by each operator and (2) the specification of the application as a composition of operators. Figure 1 shows the Lagniappe application meta-model.

2.1.1 State Specification

Most packet processing applications maintain persistent state for some flow definitions. Hence, the state specification model in Lagniappe consists of two parts:

- *Flow signature*: A flow is defined using a set of fields contained in a packet (generally, in the packet header). For instance, a flow may refer to all packets with the same 5-tuple value: {*sourceIP, destinationIP, sourcePort, destinationPort, protocolID*}. Thus, the State entity for each Operator instance includes a specification of the Flow Signature entity that identifies several packet Field entities used to generate flow identifiers. It is this flow signature that defines the access semantics of the state and allows Lagniappe to provide efficient state access by reducing contention for shared state.

---

[1] *Lagniappe* is a Cajun French word meaning, "something extra."

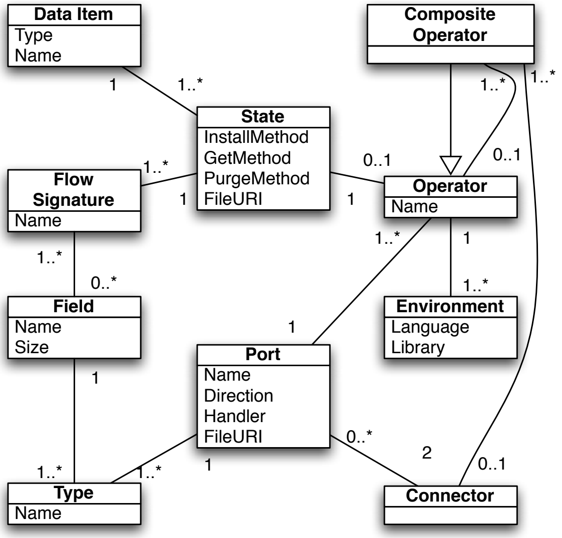

Figure 1: The Lagniappe Application Meta-Model

- *State description*: The description of the state includes the specification of each data item included in the state, as well as methods for accessing the state. Thus, the State entity for each Operator instance relates to one or more Data Item entities that define the names and types of persistent state. The State entity also has three important properties: (1) the *Install* property identifies a function that, given state and a flow identifier, adds the state to the operator's internal data structure, (2) the *Purge* property identifies a function that deletes state from the internal data structure corresponding to a flow identifier, and (3) the *Get* property identifies a function that provides access to state corresponding to a flow identifier. These state maintenance methods must protect state access (much like the procedural operator code).

2.1.2 Specifying the Graph

Lagniappe structures packet processing applications as directed graphs of independent operators. Operator is the primary entity in the application meta-model; it represents a collection of packet processing functions. Programmers refer to Operator entities by their *Name* property. Each Operator is associated with two basic entities: (1) Environment and (2) Port. The Environment entity identifies the language, compiler, and runtime environment required to execute the operator; the Port entity denotes the connectivity points of an Operator. The *Direction* property defines the Port as either incoming or outgoing. For incoming ports, the *Handler* property identifies the function Lagniappe invokes to process a packet. The the Type entity specifies the format of the Port.

Lagniappe allows specification of application graphs by interconnecting Operator instances through their Port instances. The connectivity is specified using the Connector entity. The Composite Operator entity allows programmers to define connected sub-graphs of Operator entities as a module that behaves as an Operator, allowing programmers to compose and reuse code easily.

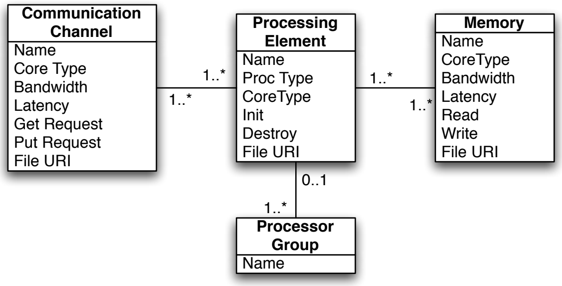

Figure 2: The Lagniappe System Meta-Model

2.2 System Meta-Model

The system meta-model specifies the number and types of processing elements available in the system, as well as methods for communicating (using memory or message passing channels) across processing elements. Figure 2 shows the Lagniappe system meta-model.

The Processing Element entity is the core of the system meta-model. The *Name* property gives each Processing Element instance a unique identifier, and the *Type* property defines the type of the element (e.g., a Pentium or a SPARC core). The *Init* and *Destroy* properties identify functions to initialize or destroy a processing element. Each Processing Element can be a member of a Processor Group that signifies a separate executable is needed as these processors are in a separate system. The Processing Element entities define the set of processing resources in the system to which Lagniappe can schedule operators to provide automatic mapping.

Each Processing Element can communicate with other Processing Element entities either using shared memory or through a message passing channel. A Communication Channel entity represents a message passing mechanism across processing elements. The *Name*, *Bandwidth* and *Latency* properties specify some basic features of a channel. The *GetRequest* and *PutRequest* properties define the methods used to interact with the channel. The *FileURI* property identifies the location for the drivers for all of these functions. The Communication Channel entity allows Lagniappe to monitor processor load and thus handle changes in workload by triggering an appropriate adaptation policy when a processor is overloaded or under utilized.

The Memory entity is similar to the Communication Channel entity. The *Name*, *Bandwidth* and *Latency* properties specify basic features of a memory level, while *Read* and *Write* refer to the methods used for accessing the memory level.

It is important to observe that the Lagniappe system meta-model does not include any aspects of the system that pertain to the execution of an operator on a single processing element. The system model only describes the "multi" aspects of the multi-⋆ platform. We assume that the tasks of compiling and executing applications onto a single processing element is handled by the compiler and run-time system available for that element type.

3 Programming Environment

The Lagniappe system architecture is composed of two independent compilation paths. The application and system model compilers are developed using the ANTLR language translator tool [25]. The model compilers take as inputs the application and system models specified in XML format and then generate C++ code. The generated C++ code defines classes for the system and application entities that are derivatives of the platform-independent *Lagniappe system library*.

3.1 Lagniappe Library

The Lagniappe system library contains 6 major classes. The `Application` and `System` classes act as the containers for the `Operator`, `ProcElement`, `CommChannel`, and `Memory` classes. All these classes are abstract and are instantiated by the compilers.

The compiler generates wrapper classes around the system-specific resource implementations that the system programmer provides as well as classes representing the different operators in the application. While the generated code from both sides—the application and the system—interfaces with the Lagniappe library, the application programmer never directly interfaces with the system specific code. The Lagniappe library acts as an intermediary between the platform-independent code that the application compiler generates, and the system-specific code the system compiler generates. The Lagniappe library also contains the logic to schedule operators to resources, to monitor channels and detect overloaded resources, and to adapt the assignment of resources when the workload changes.

3.2 Application Compiler

The application compiler generates classes derived from `Operator` as well as an instance of `Application` that is specific to the particular application model:

1. For each Operator entity in the application model, a new class is generated that is derived from `Operator`.
2. If the Operator relates to a State entity, the persistent state is declared as a private member variable of the `Operator` and the state access methods are declared as private methods. The compiler generates public wrapper methods for these private ones.
3. For each Port of type *IN*, the associated *Handler* is declared as a private method. For each Port of type *OUT*, a private method is declared with the *Name* of the port. The application programmer uses this method to send requests from respective port.
4. If the State relates to a Flow Signature entity, that Flow Signature is used by the compiler to define a `getFlowId` private method that is used for flow-based load balancing during runtime.
5. The compiler creates a class that is inherited from the `Application` class that implements its two major abstract methods: `buildOperators` and `connectGraph`.
6. Lastly, the application compiler creates the main application file that creates an instance of both the generated `Application` class and the `System` class. It calls `createResources`, `buildOperators`, and `connectGraph`. Finally, the `schedule` method of the application is called on the system object.

3.3 System Compiler

The system compiler generates classes derived from `ProcElement`, `CommChannel`, and `Memory`. The compiler also generates an instance of `System` specific to the system model:

1. For each Processing Element, Communication Channel, and Memory entity a new class is defined that is derived from `ProcElement`, `CommChannel`, and `Memory`, respectively. A private member variable is declared of *Core Type*. The classes' respective abstract methods are instantiated. For the classes derived from `Memory` and `CommChannel`, the *Bandwidth* and *Latency* values are stored as constants within the generated classes.
2. Lastly, an instance of the `System` class is generated. The abstract method `createResources` is implemented. First, `Memory` and `CommChannel` classes are generated for each model instance. Then, the compiler uses the entity relationships to define the connectivity of the `ProcElement` instances.

3.4 Benefits

The generated code addresses the three primary challenges—application to resource mapping, state management and workload distribution, as well as dynamic adaptation of resource allocations—in using multi-$\star$ platforms to design high-throughput packet processing applications.

- *Resource Mapping*: In Lagniappe, operators are never split across processing elements; more than one operator can be assigned to a single processing element (based on the interconnections specified in the application model). Using the hybrid mapping model, a packet may traverse multiple processing elements during its lifetime, and each application stage may be replicated. Based on workload fluctuations, Lagniappe determines at run-time the number of replicas of each operator. The thread-safe requirement on operator code ensures correctness of operation even when the system adjusts the number of replicas over time.
- *State Management and Load Distribution*: Lagniappe compilers utilize the persistent state definitions for each operator to generate a custom load distributor per operator. In particular, the compiler generates code to

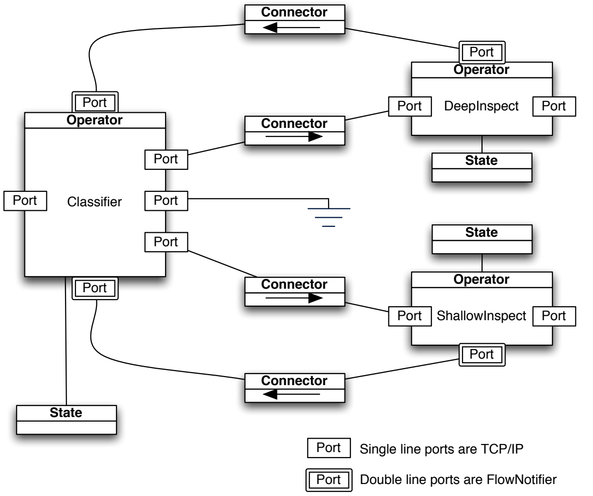

Figure 3: The Lagniappe model of a two-level malicious flow detector

distribute load across replicas of an operator using the flow definitions provided in the operator models. The load distributor pins each flow to a particular replica and re-pins the flow upon detecting any overload at the replica. When a flow is re-pinned, the state access methods specified with the operator migrate and install the state of a flow at a different replica.

- *Resource Adaptation*: The processing resources required to execute operators may change with fluctuations in workload. Lagniappe generates code to monitor each channel that communicates packets across processing elements. Lagniappe monitors each processing element's incoming communication channel to determine when the workload exceeds the processor's capacity. In the case of excess workload, the monitors triggers resource allocation adaptation. The adaptation policy determines the new resource allocation (e.g., by adding additional processing capacity to handle part of the workload from the backlogged element). The Lagniappe run-time system (1) uses the state management methods specified for the operator(s) running on the backlogged element to migrate and install state at the newly added processing element and (2) adjusts the load distribution to include the newly added processing element. If a queue becomes empty, Lagniappe invokes the adaptation policy to determine if the processing element should be deallocated.

4 Lagniappe Examples

4.1 Application

We present the Lagniappe model of a two-level malicious flow[2] detector in Figure 3. Initially, the application directs all incoming traffic to the *ShallowInspect* operator, where a lightweight detection algorithm is run on the packet flow. If a flow is flagged as a possible malicious flow, a special control message is sent to the *Classifier*. The *Classifier* operator's flow tables then are updated to send all subsequent packets of the suspected flow to the *Deep Inspection* operator. If the *Deep Inspection* operator determines the flow is indeed malicious, a control message is sent to the *Classifier*, and all subsequent packets are dropped.

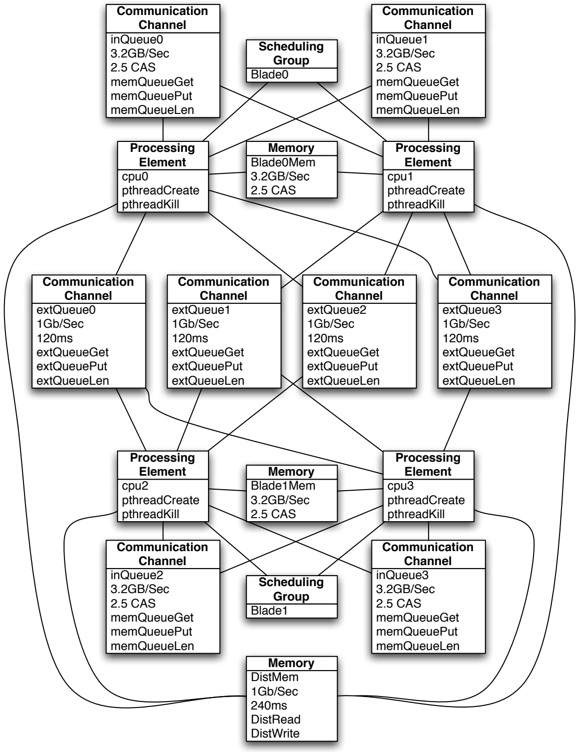

Figure 4: The Lagniappe model of a two blade system. Each blade contains two processors.

4.2 Platform

Figure 4 shows the Lagniappe model of a four-way blade system comprises two blades of two processors each. The processors within a blade can deliver requests to each other using fast, main-memory-based communication channel implementations. To send requests across blades, a slower communication channel (possibly using the blade server's backplane) must be used. The processors of each blade are in a scheduling group; each blade needs its own executable application. Also, if persistent state must be transferred across blades, the software distributed shared memory is used.

5 Related Work

Related packet processing programming environments can be broken into two major groups:

High-Level Environments Click [13] is the most well-known packet processing programming environment. Click

---

[2] Malicious could mean a virus, a worm, or an intrusion attempt. The basic architecture would be the same for any of these applications.

allows programmers to specify applications in terms of a connected graph of independent operators, but it was written for a single thread. Follow-on work with MPClick [5] expands Click to utilize multiple threads. Click, however, has no mechanisms for adapting resource allocation. Thus, Click does not handle major changes in workload that cannot be dealt with minor overprovisioning. Click is written as a Linux module, with no real way to separate a Click application from the underlying Linux platform. As well, there is no inherent support for processor heterogeneity beyond what Linux provides.

A more recent system is Aspen [20]. Aspen does not address the main issues that make multi-$\star$ development difficult, namely state access, in the language design. While runtime adaptation is supported, nothing is done to guarantee efficient access to persistent state while balancing load among resources.

Low-Level Environments NesC [8] is a low-level dialect of C created to specifically deal with the embedded restrictions of sensor networks; and thus, it does not provide enough flexibility. Nova [9] is a language specifically designed to be easy to compile for the IXP, where hardware details are exposed explicitly to the programmer. Baker [4], while presenting a high-level programming environment, is designed specifically to compile to Intel's IXP and exposes the platform details to the programmer.

6 Conclusion

In this paper, we describe the design of Lagniappe, a programming environment that simplifies the design of portable, high-throughput packet-processing applications on multi-$\star$ architectures. Lagniappe uses a hybrid programming model: It combines a procedural specification of the basic operators for processing packets with a model-driven declarative specification of the various features of the operators and the target hardware platform. Using the declarative specification, the Lagniappe programming environment automates the mapping of applications onto the multi-$\star$ platform, ensures efficient and coherent accesses to persistent, shared state, and performs dynamic allocation of resources to operators.

References

[1] J. Andrews and N. Baker. XBox 360 System Architecture. *IEEE MICRO* 26(2), 2006.

[2] J. R. von Behren, J. Condit, F. Zhou, G. C. Necula and E. A. Brewer. Capriccio: Scalable Threads for Internet Services. In Proc. of *Symposium on Operating Systems Principles*, Bolton Landing, New York, October 2003.

[3] B. Burns, K. Grimaldi, A. Kostadinov, E. Berger and M. Corner. Flux: A Language for Programming High-Performance Servers. In Proc. of *USENIX Annual Technical Conference*, Boston, Massachusetts, May 2006.

[4] M. K. Chen, X. Li, R. Lian, J. H. Lin, L. Liu, T. Liu and R. Ju. Shangri-La: Achieving High Performance from Compiled Network Applications While Enabling Ease of Programming. In Proc. of *SIGPLAN Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.

[5] B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In Proc. of *USENIX Annual Technical Conference*, Boston, Massachusetts, June 2001.

[6] W. Eatherton. The Push of Network Processing to the Top of the Pyramid. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Princeton, New Jersey, October 2005.

[7] P. Emma. The End of Scaling? Revolutions in Technology and Microarchitecture as We Pass the 90 Nanometer Node. In Proc. of *International Symposium on Computer Architecture*, Boston, Massachusetts, June 2006.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler. The *nesC* Language: A Holistic Approach to Networked Embedded Systems. In Proc. of *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[9] L. George and M. Blume. Taming the IXP Network Processor. In Proc. of *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2003.

[10] W. L. Hürsch and C. V. Lopes. Separation of Concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts, February 1995.

[11] P. Hofstee. Power Efficient Processor Architecture and the Cell Processor. In Proc. of *International Symposium on High Performance Computer Architecture*, San Francisco, California, February 2005.

[12] A. Kleppe, J. Warmer and W. Bast. MDA Explained. Addison-Wesley, Boston, Massachusetts, 2003.

[13] E. Kohler, R. Morris, B. Chen, J. Jannotti and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems* 18(3):263–297, August 2000.

[14] R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason and H. . Vin. A Case for Run-Time Adaptation in Packet Processing Systems. In Proc. of *Workshop on Hot Topics in Networks*, Boston, Massachusetts, November 2003.

[15] P. Kongetira, K. Aingaran and K. Olukotun. Niagara: A 32-Way Multithreaded Sparc Processor. *IEEE MICRO* 25(2):21–29, 2005.

[16] P. Li and S. Zdancewic. Combining Events and Threads for Scalable Network Services. In Proc. of *SIGPLAN Conference on Programming Language Design and Implementation*, San Diego, California, June 2007.

[17] M. Macedonia. The GPU Enters Computing's Mainstream. *IEEE Computer* 36(10):106–108, 2003.

[18] T. L. Riché, J. Mudigonda and H. M. Vin. Experimental Evaluation of Load Balancers in Packet Processing Systems. In Proc. of *Workshop on Building Block Engine Architectures for Computers and Networks*, Boston, MA, October 2004.

[19] J. S. Turner. A Proposed Architecture for the GENI Backbone Platform. In Proc. of *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, San Jose, California, October 2006.

[20] G. Upadhyaya, V. S. Pai and S. P. Midkiff. Expressing and Exploiting Concurrency in Networked Applications in Aspen. In Proc. of *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Jose, California, March 2007.

[21] M. Welsh, D. E. Culler and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Proc. of *Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[22] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org/.

[23] Intel IXP Family of Network Processors. http://www.intel.com/design/network/products/npfamily/.

[24] Intel Multi-Core: An Overview. http://www.intel.com/multicore/overview.htm.

[25] ANTLR, Another Tool for Language Recognition. http://www.antlr.org.