to appear, MODELS 2010

Transformation-Based Parallelization of Request-Processing Applications

Taylor L. Riché[1], Harrick M. Vin[2], and Don Batory[1]

[1] Department of Computer Science, The University of Texas at Austin, Austin, TX, USA
{riche,batory}@cs.utexas.edu
[2] Tata Research and Development Center, Pune, India
Harrick.Vin@tcs.com

Abstract. Multicore, multithreaded processors are rapidly becoming the platform of choice for high-throughput *request-processing applications (RPAs)*. We refer to this class of modern parallel platforms as *multi-⋆ systems*. In this paper, we describe the design and implementation of *Lagniappe*, a translator that simplifies RPA development by transforming portable models of RPAs to high-throughput multi-⋆ executables. We demonstrate Lagniappe's effectiveness with a pair of stateful RPA case studies.

1 Introduction

Moore's law and the accompanying improvements in fabrication technologies (90*nm* to 65*nm* and beyond [7]) have significantly increased the number of transistors available to processor designers. Rather than use these transistors by further extending the depth of processor pipelines, designers are developing processors with multiple, multithreaded cores. This class of modern parallel platforms, called *multi-⋆ systems*, are the standard for constructing high-performance and high-throughput computing clusters.

Sadly, the ubiquity of multi-⋆ systems has not led to breakthroughs in easing the core burdens of parallel programming:

1. It is the responsibility of programmers to map components of an application to available parallel resources manually, in order to maximize performance or efficiency. Further, the computational needs of an application may change during runtime, requiring mapping decisions to change dynamically based on both application and workload characteristics [17].
2. Many applications have persistent, mutable state. Accessing state with multiple execution threads, if done naïvely, can cause contention for locks and reduce performance [23]. As multi-⋆ systems grow larger, application mappings become more difficult as the cost of sharing state between processing elements grows.
3. Different systems utilize different types of parallel resources. Platform-specific code should never be embedded in an application because the application may run on different hardware configurations over time (*portability for code reuse*) and because multi-⋆ systems are often heterogeneous comprising different hardware types (*portability for heterogeneity*).

Request-processing applications (RPAs) are an important class of applications on multi-⋆ systems given their traditionally high amounts of both task parallelism (i.e. different parts of the application can run in parallel [31]) and data parallelism (i.e. the same part of the application can process different requests at the same time [30]). RPAs are a directed graph of components called streaming [28] or pipe-and-filter [2,25] architectures. Multi-⋆ concerns of these architectures (e.g. component coordination, access to persistent state, and mapping the application to multi-⋆ hardware) are non-trivial. Even worse, programmers implement RPAs using traditional, low-level methodologies that strain development, testing, component reuse, and maintenance. While tools [1,6,11,16,28,29] do allow programmers to create high-performance applications—even parallel ones—the programming task remains daunting and error-prone.

Our research improves the state-of-the-art tools in RPA development for multi-⋆ systems by automating the difficult, tedious, and error-prone tasks that lie at the heart of parallel RPA programming. We built *Lagniappe*[3], a translator that maps a model of an RPA to a C++ module; effectively it implements a sophisticated *model-to-text (M2T)* transformation. Lagniappe was our first step in understanding the domain of RPAs and its engineering requirements. It helped us to test the correctness of our RPA abstractions and demonstrated that we could derive efficient multi-⋆ implementations of RPAs from these abstractions. But we now see Lagniappe as a precursor to a more sophisticated *model driven engineering (MDE)* tool where a model of an RPA is progressively mapped to more complex models using *model-to-model (M2M)* transformations that expose task and data parallelism. The most refined model is then translated to a C++ module via a simple M2T transformation. Parallelism mappings are present in Lagniappe now, but are hardcoded in its M2T implementation.

In this paper, we present the design and implementation of Lagniappe from this more advanced MDE perspective. An RPA is specified by a graph of interacting components where each component is implemented in an imperative language (C++ in our case). The model captures application features that are relevant for execution on multi-⋆ systems. Lagniappe automatically transforms this model of an RPA—actually a single-⋆ description that can only execute on a single processing element (processor, core, thread, etc.[4])—into a model that is optimized for multi-⋆ execution. These transformations determine custom and optimized policies for run-time adaptation and load-distribution. Lagniappe automatically creates an application that meets the four challenges of multi-⋆ programming: dynamic resource assignment, efficient state access, portability, and parallelism. We demonstrate experimentally the effectiveness of Lagniappe's mappings with two stateful RPA case studies.

We begin with a review of the key traits of the RPA domain, followed by an MDE-description of how Lagniappe parallelizes RPAs by transformations.

---

[3] Lagniappe is a Cajun-French word meaning "a little something extra." If programmers give us a small amount of extra information, we can give them much added benefit.

[4] When we refer to a processor we are referring to one chip. A processor may have several cores, a core being a fully functional processing element with its own low-level cache. Operating systems typically see all cores as separate "processors."

2 Request-Processing Application Domain

2.1 General Structure

An RPA is specified by a directed graph of *A_Operators* (also known as components, filters, and stages [16,25,27]). Each A_Operator may contain multiple inputs and outputs, and may generate zero, one, or multiple requests when processing an incoming request. Requests flow through *A_Connectors* between A_Operators. A_Connectors are typed by the messages they carry and *Ports* are typed by the messages they send or receive. A request enters the application at the head of the graph and passes from one A_Operator to the next until an A_Operator either drops the request (i.e. the system reclaims the resources it assigned to request) or the request exits the graph.

Figure 1 depicts a model of an RPA we call the *AttackDetector* that examines incoming requests for patterns that would suggest a network attack.[5] The *AttackDetector* has three A_Operators: *Classifier*, *Inspect*, and *Service*.

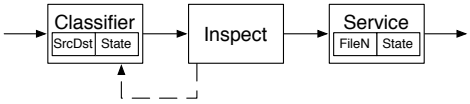


Fig. 1. The *AttackDetector* Application

The behavior of *AttackDetector* is straightforward: requests enter *Classifier* and are marked as either good or bad. Bad requests are dropped immediately. Good requests move to *Inspect* where it performs a test to determine if the request is part of an attack. If the request is not an attack, *Inspect* forwards the request untouched to *Service*. If *Inspect* thinks the request may be part of an attack, the request is dropped and *Classifier* is notified to drop all future related requests. Requests that pass *Inspect* are processed by *Service* which represents any service, such as a webserver or database engine. *AttackDetector* processes two different types of requests: dotted lines represent notifications of suspicious activities and solid lines are normal paths of requests.

2.2 Request Flows and Flow Signatures

The processing of a request may have no impact on processing other requests in a system. At other times, A_Operators may process related requests. A *flow* is any sequence of requests (not necessarily adjacent in arrival time) that share some property, for example a source and destination network address pair. A *flow signature* is a function that maps a request to a *flow identifier*. All requests that map to the same flow identifier are said to be in the same flow.

An application may exploit several different flow signatures. In *AttackDetector*, there are two flow signatures: *SrcDst* and *FileN*. *Classifier* uses (source, destination) address pairs to identify flows while *Service* uses file names contained in the request.

[5] By attack we could mean worm propagation, distributed denial of service, or simple "hacking." The exact definition is not important for this example.

2.3 Persistent A_Operator State

Unlike classic pipe-and-filter architectures where filters are stateless [25], many request-processing applications maintain persistent state—state that exists beyond the processing of any one request. Accesses to persistent state can dominate the time it takes an A_Operator to process a request [20] and naïve concurrent access can introduce contention for locks [23]. Persistent state is essential in many RPAs because the logic of an A_Operator may depend on the requests it has previously processed.

Persistent state falls into two classes. The first assumes no relationship between requests. The second uses flow identifiers as keys to data associated with a flow, which we refer to as *flow state*. In $AttackDetector$, $Classifier$ maintains a mapping of flow identifiers to state representing the status (good/bad) of the flow. *Service* contains application state (such as webpages) and uses a different flow identifier.

2.4 Application Guarantees

Application designers create RPAs to meet specific throughput or latency guarantees. Such guarantees are often defined by a business-driven *service level agreement (SLA)*. Programmers may not care to maximize performance (e.g. maximize throughput or minimize latency) but rather to just meet the throughput or delay guarantees of an SLA. For example, a company may sell different levels of service. Different applications running "in the cloud" may receive different amounts of resources and achieve different levels of performance based on the fee paid by their customers.

3 Model and Transformations of RPAs

Lagniappe identifies two types of parallelism in an application: *task parallelism* (breaking the application into pipeline stages) and *data parallelism* (replicating a stage during runtime to process different requests in parallel). Programmers specify an RPA as a single-$\star$ model that conforms to the basic application structure presented in Section 2 with no knowledge of underlying multi-$\star$ hardware. That is, the model in Figure 1 conforms to a domain defined by a metamodel A ("A" being short for *Application*). Two other models are used as input to Lagniappe: W is the metamodel of typical A_Operator workloads, and M is the metamodel of multi-$\star$ systems. The input to Lagniappe is a model from each of these three domains.

The Lagniappe translator is depicted in Figure 2. Lagniappe is currently a pair of M2T transformations $A2C : A,W,M \rightarrow C$, which maps a ($A$,$W$,$M$) triple to a C++ module, and $M2C : M \rightarrow C$, which maps M to another C++ module. In the next sections, we offer a more advanced description of Lagniappe by decomposing the $A2C$ transformation as a series of M2M transformations ($REP \bullet PIPE \bullet PROF$) followed by a simple M2T transformation $R2C$. That is, $A2C = R2C \bullet REP \bullet PIPE \bullet PROF$. We explain our work using metamodel instances. The details of our metamodels are given in [24].

3.1 A: The Single-$\star$ Application Model

Figure 1 is a model of $AttackDetector$ that conforms to A. Beyond what we presented earlier, $Classifier$ is stateful and uses the Flow Signature *SrcDest* to map requests to

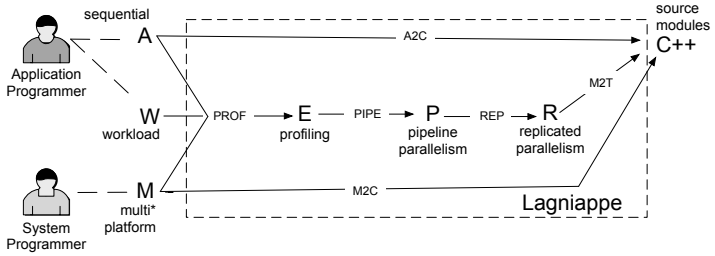


Fig. 2. The Lagniappe Translator

flow identifiers. *Service* is also stateful and uses the Flow Signature *FileN* to map the names of files contained in the request payload to a flow identifier. The input/output ports of all A_Operators have the data-request type except for the A_Connector from *Inspect* to *Classifier* that reports attacking flows (the dashed arrow in Figure 1). *A key property of an A model is that no two A_Operators can access the same persistent state; doing so forces programmers to model task parallelism in the application as each A_Operator can now run independently with no contention for shared state.*

3.2 Performance Estimates (*PROF* Transformation)

The $PROF : A, W, M \rightarrow E$ transformation uses the application, workload, and platform models to produce an annotated application model in domain E (for *Estimate*). Metamodel E is a minimal extension to A: E preserves all of the objects and relationships in A, and adds one attribute to each A_Operator to provide elementary performance information. The programmer provides an average case workload (W) for each A_Operator and Lagniappe determines the A_Operator's average execution time on platform M through profiling. These estimates are added to the A model to produce an E model and are used in the transformations that we discuss next.

3.3 Task Parallelism (*PIPE* Transformation)

Lagniappe parallelizes an application in two ways. The first identifies *pipeline stages* (i.e. task-parallel computations). We call these pipeline stages *P_Operators* for pipelined A_Operators. One or more A_Operators are contained in a P_Operator.

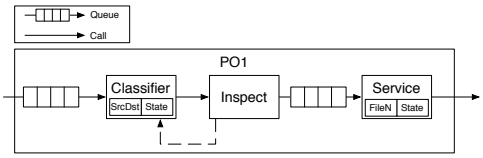


Fig. 3. *P* Model of *AttackDetector*

A_Connectors are mapped to *P_Connectors*, which are one of two types: *Call* or *Queue*. An A_Connector is mapped to a Call if it is internal to a P_Operator. A Call implements a P_Connector by method calls. All other P_Connectors are Queues. Queues provide an asynchronous FIFO abstraction between P_Operators.

P is the metamodel that defines the domain of pipelined applications using P_Operators and P_Connectors. Figure 3 shows the result of transforming the *AttackDetector* in Figure 1 to its *P* model counterpart (note that the performance information of E is not shown). We refer to this transformation as $PIPE : E \rightarrow P$.

Policy Discussion. A first thought is simply to use A_Operators as pipeline stages and not go through the trouble of forming P_Operators. While pipelining has its benefits, it also comes with overhead. Every time a request moves from one processing element to another it has to be enqueued and dequeued. As queueing overheads can be significant, it is important to keep the number of pipeline stages to a minimum. Given an equal number of parallel processing elements, we would rather have fewer pipeline stages with more chances to replicate stages than to execute separate pipeline stages on their own. This minimizes the overhead per request and provides more replication opportunities. We show in Section 4.2 that these are both desirable properties. With this in mind, we greedily form P_Operators and minimize the number of P_Operators to maximize replication possibilities.

Situations exist where an application may have A_Operators that cannot be replicated for correctness reasons (the *FT* application in Section 4.1 is one). Thus, we allow application programmers to mark A_Operators with a "Do Not Replicate" flag in *A* models, information that is preserved in *PROF* and *PIPE* transformations. If *PIPE* encounters an A_Operator with this flag, it is put into its own P_Operator. While we cannot take advantage of data parallelism by replicating such a P_Operator, we can take advantage of task parallelism by running these "Do Not Replicate" P_Operators on different processors at the same time. We show in Section 4.2 pipelining the application in this situation improves throughput.

Flow-Based Composite Policy. The formation of P_Operators proceeds as follows:

1. Start at incoming request sources.
2. Absorb A_Operators into a P_Operator along the application graph until an A_Operator with the "Do Not Replicate" flag is found or the only available output Port has a different data type than the input.[6]
3. Do not absorb A_Operators that have already been included in a P_Operator (i.e. check for cycles).
4. After a depth first search is finished, collect remaining A_Operators into P_Operators based on flow signatures. These are A_Operators that process control data and are most likely off of the application fast-path. Combining them usually does not impact performance and saves system resources (i.e. a smaller number of overall pipeline stages equals a smaller number of processing resources initially needed).

Applying the above heuristic to Figure 1 produces one P_Operator, shown in Figure 3. That is, *Classifier*, *Inspect*, and *Service* are put into P_Operator *PO*1. The incoming A_Connector for the P_Operator *PO*1 is now a Queue. All of the A_Connectors internal to *PO*1 are Calls.

[6] A Port with a different data type implies a control-data path through the application, and one that will not be externally visible.

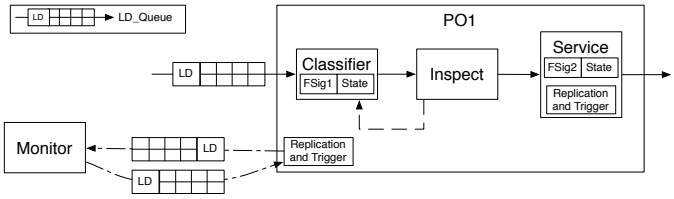


Fig. 4. *R* Model of *AttackDetector*

We could have used other policies. For example, memory may be limited in embedded platforms. Any one P_Operator may have an upper bound on its code size [4]. For the class of RPAs that motivated our work, code size was not an issue and flow state was prevalent. If we needed to support another pipelining policy i, we would implement a different transformation $PIPE_i : E \rightarrow P$. More on this later.

3.4 Data Parallelism (*REP* Transformation)

Lagniappe's second parallelizing transformation takes advantage of data parallelism in an application. Individual pipeline stages are transformed to support processing multiple requests at the same time. Let R be the metamodel that defines the domain of applications that support replication and let transformation $REP : P \rightarrow R$ map a P representation of an application to an R representation. The first step in REP is to transform P_Operators into R_Operators.

An *R_Operator* contains platform-independent implementations of two mechanisms. The first is the data structures that map R_Operator replicas to individual processing elements. The second is an adaptation trigger that requests more processing resources from the system when an R_Operator becomes overwhelmed by its current workload. The processing element mapping is, at this point, underspecified as we have not yet finished pairing the application with a specific multi-$\star$ platform and we do not replicate any R_Operators until the workload requires it at runtime. We use the execution time annotations in E (which have been copied into P) to create an adaptation trigger implementation; we discuss our specific policy later in this section.

The next step in REP is to replace all Queues in the application with *LD_Queues* (the "LD" stands for *load distributing*). LD_Queues implement a queue abstraction that distributes incoming requests amongst the replicas of an R_Operator according to a policy (we use a policy based on flows and discuss alternatives later in this section).

The REP transformation is completed with the addition of a special R_Operator ($Monitor$) to the application graph. $Monitor$ tracks resource usage across the application and makes decisions to allocate more processing elements to R_Operators that request more resources. The $Monitor$ is also underspecified because the amount of available resources is not yet known.

Figure 4 shows the *AttackDetector* after REP is applied. REP maps $PO1$ to an R_Operator. Queues are replaced by LD_Queues and Rep adds the $Monitor$ R_Operator. Bidirectional communication is added between $Monitor$ and $PO1$.

Policy Discussion. LD_Queues distribute load among replicas using a policy based on flow signatures of their respective R_Operators. Specifically, we pin flows to replicas. When a request is enqueued, an LD_Queue uses the flow signature to generate a flow identifier. The LD_Queue sends all requests from a flow to the same replica. *The value of flow pinning is performance: load distribution policies are based on flow signatures. Flow pinning guarantees that all lock acquisitions based on flow identifiers will never block, thereby increasing RPA performance.*[7]

If an R_Operator has no flow signature attached to it, then the LD_Queue uses a simple round-robin scheme for load-distribution. If an R_Operator comprises multiple P_Operators then we choose the Flow Signature from the A_Operator with the largest execution time. The reasoning here is that using that Flow Signature will reduce contention for shared state the most within the R_Operator if the R_Operator is replicated during runtime.

REP also adds an adaptation trigger to R_Operators. This trigger uses the policy of Kokku et al. [18]. The application has a maximum latency value that all requests must observe. This latency is specified by an SLA. We split that latency amongst the R_Operators and determine the maximum amount of time a request can spend in an LD_Queue and not violate the delay guarantee. We translate this *slack* value to queue depth using Kokku et al.'s methodology. If the queue depth violates this determined value, the R_Operator requests more resources.

An LD_Queue repins flows when the number of processing elements assigned to a R_Operator changes. Our case studies have long-lived flows, so repinning was beneficial. Other polices do exist for load distribution, and are dependent on workload [8,9] and system properties [23].

Lagniappe presently hard-wires the above policy to implement $REP : P \rightarrow R$. A different policy j would be implemented by different transformation, $REP_j : P \rightarrow R$. Choosing the best *REP* transformation is discussed later.

3.5 Code Generation (*R2C* Transformation)

Lastly, Lagniappe performs a simple M2T transformation to map an *R* model to a C++ module that is platform-independent. The binding of the application code to a particular multi-$\star$ system implementation is done at runtime (which we discuss in Section 3.7). The Lagniappe library allows us to generate platform-independent code as it acts as an adaptor between the generated C++ module and the multi-$\star$ system.

3.6 System Models

Let *M* (for multi-$\star$ system) denote the domain/metamodel of system models that support fully parallel (pipelined and replicated) applications. An *M* model consists of four entities: Elements, Channels, Mutexes, and Groups. *Elements*, short for processing elements, represent an abstraction that can support independent computation. They can be

[7] When locks are acquired based on flow identifiers, and all requests from the same flow go to the same replica, then two requests from the same flow will never be processed simultaneously guaranteeing a replica will never block waiting to acquire a lock.

software threads (such as classic UNIX pthreads) or actual hardware contexts (such as hardware threads of an embedded system). *Channels* allow data to flow between Elements. A Channel must support multiple queues per Element to allow load sharing between R_Operators (and the separate incoming ports of R_Operators) assigned to the Element. *Mutexes* provide a way to ensure consistency when multiple Elements execute a replicated R_Operator at the same time. Finally, *Groups* signify relative data sharing costs amongst Elements (for example, the cores of one physical CPU are in a Group, but the cores of two separate physical CPUs are not).

The Lagniappe translator maps an M model to a C++ module by the M2T transformation $M2C : M \rightarrow C$. By separating the modeling of the multi-$\star$ systems (M) from that of the RPAs (A), RPA models are portable because they do not have to be modified for different multi-$\star$ systems. With changes to M models, we have run Lagniappe applications on Linux, Mac OS X, and OpenSolaris.

3.7 Run-Time Binding of Applications to Platforms

To recap, Lagniappe uses transformation $A2C$ to map a model a of an RPA to a C++ module m_a and transformation $M2C$ to map model m of a multi-$\star$ system to another C++ module m_m. Once m_a and m_m are compiled, they are linked with a module m_{bind} in the Lagniappe run-time library to produce a multi-$\star$ executable of a.

The library module m_{bind} defines a generic way to map R_Operators and LD_Queues to multi-$\star$ Elements and Channels at run-time. This mapping has two parts:

1. The *Monitor* R_Operator now has a complete list of system resources and knows which Elements have been assigned to which R_Operators and which Elements are free to assign during runtime when an R_Operator requests more resources.
2. Locking calls in the R_Operator modules are mapped to Mutex implementations.

4 Case Studies

We use two different applications to show the versatility and performance of Lagniappe: 1) a concrete version of our running *AttackDetector* example and 2) a fault-tolerant replicated-state machine, called FT. *AttactDector* is now familiar; we next discuss FT.

4.1 The FT Application

FT is the part of a replicated-state machine that decides the order in which requests are processed by a server (such as our *AttackDetector* application). We reengineered a hand-coded state-of-the-art prototype of a replicated-state machine for fault-tolerant servers [5]. Our motivation to do so was to show that Lagniappe is suitable for complex applications as well as simple ones. Figure 5 shows a model of the FT application.

FT attaches to three separate networks: the clients submitting requests, the servers that execute requests, and copies of FT on different machines that together decide upon the execution order of requests. FT is part of a much larger distributed-application running across many machines that make up the abstraction of one always available and

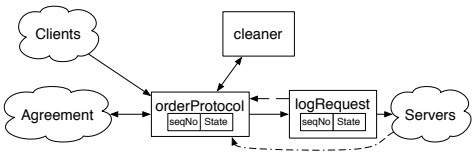


Fig. 5. The application model for the FT application.

correct server. Clement, et al. give a full description of their version of the application [5]; we describe its key details below.

Requests from clients arrive at the *orderProtocol* A_Operator, where they are sent to the *cleaner* to have their digital signatures verified. If the signature is valid, the request is returned to *orderProtocol*.[8] *orderProtocol* then initiates the actual agreement protocol with FT copies on other machines. The application copies run a three-phase protocol where the next message to be processed is proposed, accepted as a valid proposal, and then voted upon. If enough FT copies agree, the message is marked as the next message to be executed.

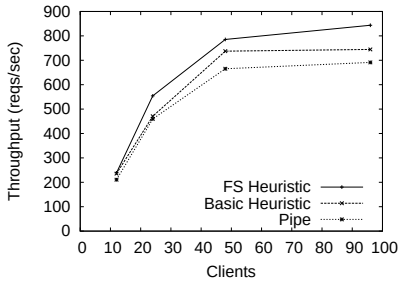
orderProtocol then sends the request to the *logRequest* where the request is logged to stable storage for purposes of failure recovery and asynchronous network support. After the write completes, *logRequest* sends the request onto the network to execution servers that process the request. The *logRequest* also alerts the *orderProtocol* that the logging has completed to disk.

Both the *orderProtocol* and the *logRequest* A_Operators have persistent state, and share a flow identifier that is the sequence number of the incoming requests. Different from *AttackDetector*, FT does not have multiple flows in flight at the same time. The entire purpose of the FT application is to serialize incoming requests from multiple clients. FT assigns each request a sequence number that denotes that request's place in the order. The sequence number defines the only flow. The *cleaner* A_Operator is stateless and could be replicated during runtime. Given the specific requirements of the FT application, each A_Operator in Figure 5 is transformed into its own R_Operator.

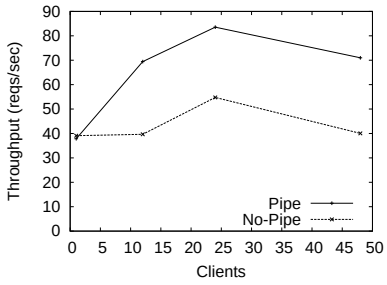
4.2 Experimental Results

Attack Detector. Figure 6(a) shows the results of scaling the number of clients for *AttackDetector* using three different policies for forming P_Operators and R_Operators. The x-axis is the number of simultaneous clients and the y-axis is average throughput. The *Service* Operator performs work to simulate a significant web real-world web service. Each client sends an HTTP request for a file and waits until it gets a response to send another request. *AttackDetector* was executed on a 16-core server running Linux 2.6.16. The clients were distributed over several 4-core servers running the same version of Linux.

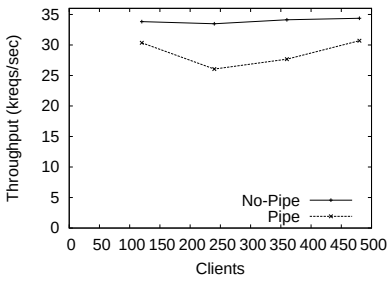
[8] While it may seem we could model the application as more of a pipeline, with the before-*cleaner* and after-*cleaner* portions of *orderProtocol* as their own A_Operators, both of these A_Operators would access the same state. As mentioned earlier, a key constraint of *A* models is that different A_Operators are not allowed to share state.



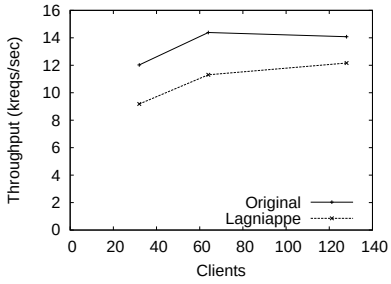
(a) Different policies for forming P_Operators in *AttackDetector*



(b) Benefits of pipelining with the "Do Not Replicate" flag set.



(c) Pipelining for very small Operators.



(d) Performance of Lagniappe FT and the original Java implementation.

Fig. 6. Experimental Results for *AttackDetector* and FT.

The results of three different policies are shown: (1) using the programmer-provided Operators as P_Operators, (2) using the P_Operator heuristic with the first Operator's Flow Signature as the load-balancing policy, and (3) our P_Operator heuristic with the most expensive Operator's Flow Signature as the load-balancing policy. The pipelined version scales the worst as it introduces latency between the stages, unnecessarily increasing the amount of time each request takes to process. Heuristic (2) scales better, but as the client load increases the lock contention in the *Service* Operator takes its toll, limiting the throughput gains from replication. Finally, the heuristic using the *Service* Flow Signature (3) scales the best as it eliminates lock contention by using flow pinning, but still replicates the execution taking more advantage of the available parallelism.

Figure 6(b) shows the benefit of pipelining an application when the Operators have the "Do Not Replicate" flag set in the application model. We change the *AttackDetector* application to have heavy-weight computation in both $Classifier$ and *Service* as well to set the "Do Not Replicate" flag in both. As the number of clients on the x-axis increases,

the pipeline version scales throughput better. This shows that when replication is not an option, pipelining can still be beneficial.

Figure 6(c) shows how the properties of the underlying multi-$\star$ system affect the choice of P_Operators during the *PIPE* transformation. We strip the A_Operators to their minimum functionality so that overhead of the Channel implementation is significantly more than any of the A_Operators. In this case, pipelining introduces more latency than that of the A_Operators, and thus a better solution is to combine all the A_Operators into a single P_Operator. Even in the situation where all A_Operators are marked "Do Not Replicate," keeping them together is the better option when the A_Operator cost is small.

FT. Figure 6(d) summarizes our experiments with the FT application. For these experiments we ran the application on an 8-core system running Linux 2.6.16. The clients and servers ran on 4-core systems running the same version of Linux. We ran three copies of the FT application, each on their own machine. The x-axis shows throughput in kreqs/s and the y-axis shows latency in ms. We scale the number of clients as we move along the line to push the systems to their maximum throughput. Lagniappe performs comparably (within 15%) to the hand-coded and highly-optimized Java version.

We posit that this slowdown is acceptable. Our version is based on a C++ translation of the original highly-tuned Java application, and the system implementation (threads, queues, etc.) in the original were written specifically for the demands of the application. These custom built mechanisms take advantage of deep application information, overwhelming any performance advantage of switching from Java to C++. By separating the application from the system code (unlike the way that the Java version is coded) it is now trivial to upgrade to more efficient queueing or processing mechanisms with the Lagniappe version.

The results here point towards us developing a next-generation MDE tool that works with Java components. An extensible, Java-based tool would allow us to use the original, supported FT implementation that is continually being updated and extended. Furthermore, a Java-based tool would make supporting the application servers in this domain (i.e. web services such as Hadoop [14]) possible as well as the predominant language of these services is Java.

5 Extensibility of Lagniappe

We initially developed Lagniappe using a single *PIPE* and *REP* transformation. Our case study experiences soon made it clear that alternatives were needed.

Instead of generalizing our existing *PIPE* transformation, bundling the decision of which $P \rightarrow R$ mapping to use *and* applying that mapping, we chose to separate these concerns into two distinct mappings: $ChoosePipe$ and $PIPE_i$. $ChoosePipe : (A, W, M) \rightarrow (A \rightarrow P)$ is a higher-order transformation that selects a particular $PIPE_i$ transformation given an application model (A), workload (W), and a multi-$\star$ system (M). If the execution time of the pipelining infrastructure (queues, load-distribution, etc.) is more than the overall execution time of the P_Operators then it is not worth forming pipeline stages, and a trivial $(A \rightarrow P)$ map suffices. However, if a system were to offer a hard-

ware queueing mechanism, pipeline stages would incur small overhead relative to even the fastest P_Operators encouraging the use of yet another $(A \rightarrow P)$ mapping.

Similar arguments apply to supporting multiple REP_j transformations and the selection of an appropriate REP_j by a $ChooseRep : (A, W, M) \rightarrow (P \rightarrow R)$ transformation. For example, different workload, system, and application characteristics affect which load-distribution policy is optimal [8,9,23]. We know that there can be many different $PIPE_i$ and REP_j transformations, and factoring out the selection of which transformation to use makes it easier to extend Lagniappe in the future.

6 Related Work

Dataflow programming can be recast in today's MDE technology as mapping models of dataflow applications to executables. While we are aware of pioneering MDE work in signal processing that is similar to RPAs [26], the most well-known work on parallelizing dataflow applications is largely outside the MDE literature [15]. Lagniappe differs from these traditional coarse-grain dataflow models as its operators do not need to wait for all its incoming ports to have data to execute. Also, RPAs can be expressed by coordination languages [22], which also can be recast in an MDE setting.

Labview [21] and Weaves [12] are among the earliest environments for programming RPAs on multi-threaded platforms (automatic pipelining and replication was not a focus of these systems). Click [16] is the most well-known packet-processing programming environment, allowing programmers to specify applications in terms of a connected graph of elements. Click supports task parallelism [3], but not data parallelism, and is written as a Linux module, tying it to a specific platform (while user-mode Click exists, programmers are forced to use the kernel for serious development).

The SAI architectural style [10] provides an elegant metamodel for RPA construction. SAI supports both persistent and volatile state, but does not provide mechanisms for creating an executable. Further, the application is taken as is, and no transformations are applied to further extract parallelism out of the application.

Flux [1] is a programming environment in which the dataflow aspects of a system are modeled separately from its request processing. Flux can use any multithreaded software library for its underlying execution. Flux has pipelining parallelism and has limited read-only replication parallelism. More recent environments, such as Aspen [29] and Merge [19], mainly focus on side-effect free applications, ignoring the difficulties that persistent state introduces to mapping and adaptation.

Stream environments (such as Streamit [28] and Streamware [13]) provide programmers with new languages that focus on stream processing. Stream programs are written in a fashion similar to standard programming, i.e., no separation between coordination and execution. Stream compilers determine how to separate the functionality of the program (usually with keyword help from the programmer) into tasks that are then scheduled on the multi-$\star$ resources of the system. This is a much more fine-grained approach to parallelism. Stream languages work well in signal-processing applications and low-level network programs that are more computationally bound and traditionally side-effect free.

Finally, the issue of extensibility is largely absent in all of the above works. Extensibility is, we believe, the key to highly-adaptable and general RPA programming tools. Lagniappe captures this extensibility in terms of higher-order transformations, selecting transformations to use based on input models.

7 Conclusions

RPAs are an important class of applications in multi-$\star$ systems. Existing tools do not adequately support the task *and* data parallelism that is inherent in RPAs. Further, some tools force programmers into the operating system kernel for serious development. The need for a better approach lead us to design and implement Lagniappe, a translator that transforms simple models of RPAs to high-throughput multi-$\star$ executables. Lagniappe addresses the major challenges of RPA programming: portability (RPA models are portable in that they do not have to be modified for different multi-$\star$ systems), efficient state access (load distribution policies are based on flow signatures, guaranteeing that all lock acquisitions based on flow identifiers will never block), dynamic resource assignment (accomplished by Lagniappe run-time libraries), and effective support for task and data parallelism.

Although the current version of Lagniappe is a sophisticated model-to-C++-text translator, we found it invaluable to explain the mappings of Lagniappe in a more advanced MDE-style which uses multiple model-to-model transformations prior to a simple model-to-C++-text transformation. Further our experience with Lagniappe reveals the importance of transformation extensibility: effective policies that map higher-level models to lower-level models may be influenced by platform, workload, and application-specific details. We have realized transformation extensibility through the use of higher-order transformations, which can select appropriate mappings to use.

Two case studies were presented that show Lagniappe's effectiveness. The first was a custom-built RPA that detects generic network attacks. It demonstrated the efficacy of our policy generation for parallelism. The second reengineered a hand-crafted state-of-the-art replicated state machine. It demonstrated Lagniappe's ability to support complicated applications with minimal performance impact.

We now understand the transformations necessary to produce high-performance, parallel RPAs automatically and mechanically. Lagniappe is our first step towards a more general MDE tool for parallelizing RPAs on multi-$\star$ systems.

Acknowledgments. We gratefully acknowledge the support of the National Science Foundation under Science of Design Grant #CCF-0724979.

References

1. B. Burns, K. Grimaldi, A. Kostadinov, E. Berger, and M. Corner. Flux: A Language for Programming High-Performance Servers. In *USENIX*, 2006.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal. *Pattern-Oriented Software Architecture*, volume 1. Wiley, 1996.
3. B. Chen and R. Morris. Flexible Control of Parallelism in a Multiprocessor PC Router. In *USENIX*, 2001.

4. M. K. Chen, X.-F. 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 *PLDI*, 2005.
5. A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. L. Riché. UpRight Cluster Services. In *SOSP*, Oct. 2009.
6. Microsoft directshow. `http://msdn.microsoft.com/en-us/library/ms783323.aspx`.
7. P. Emma. The End of Scaling? Revolutions in Technology and Microarchitecture as We Pass the 90 nm Node. In *ISCA*, 2006.
8. C. Estan and G. Varghese. New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice. *ACM ToCS*, 21(3):270–313, 2003.
9. C. Estan, G. Varghese, and M. Fisk. Bitmap Alggorithms for Counting Active Flows on High Speed Links. In *IMC*, Oct. 2003.
10. A. R. J. Francois. A Hybrid Architectural Style for Distributed Parallel Processing of Generic Data Streams. In *ICSE*, 2004.
11. 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 *PLDI*, 2003.
12. M. M. Gorlick and R. R. Razouk. Using Weaves for Software Construction and Analysis. In *ICSE*, 1991.
13. J. Gummaraju, J. Coburn, Y. Turner, and M. Rosenblum. Streamware: Programming General-Purpose Multicore Procesors Using Streams. In *ASPLOS*, pages 297–307, 2008.
14. Hadoop. `http://hadoop.apache.org/core/`.
15. W. M. Johnston, J. R. P. Hanna, and R. J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.
16. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.
17. R. Kokku, T. L. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin. A Case for Run-time Adaptation in Packet Processing Systems. In *HotNets*, 2003.
18. R. Kokku, U. Shevade, N. Shah, A. Mahimkar, T. Cho, and H. M. Vin. Processor Scheduler for Multi-Service Routers. In *IEEE RTSS*, Dec. 2006.
19. M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: A Programming Model for Heterogeneous Multi-Core Systems. In *ASPLOS*, pages 287–296, 2008.
20. J. Mudigonda. *Addressing the Memory Bottleneck in Packet Processing Systems*. PhD thesis, The University of Texas at Austin, 2005.
21. National Instruments LabView 8. `http://www.ni.com/labview/`.
22. G. A. Papadopoulos and F. Arbab. Coordination Models and Languages. *Advances in Computers*, 46, Aug. 1998.
23. T. L. Riché, J. Mudigonda, and H. M. Vin. Experimental Evaluation of Load Balancers in Packet Processing Systems. In *BEACON at ASPLOS*, Oct. 2004.
24. T. L. Riché, H. M. Vin, and D. Batory. Transformation-Based Parallelization of Request-Processing Applications. Technical Report TR-10-16, Dept. of CS, UT Austin, 2010.
25. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
26. J. Sztipanovits, G. Karsai, and T. Bapty. Self-Adaptive Software for Signal Processing. *Commun. ACM*, 41(5):66–73, 1998.
27. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, 1997.
28. W. Thies. *Lang. and Compiler Support for Stream Programs*. PhD thesis, MIT, Feb. 2009.
29. G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and Exploiting Concurrency in Networked Applications in Aspen. In *PPoPP*, 2007.
30. Data parallelism. `http://en.wikipedia.org/wiki/Data_parallelism`.
31. Task parallelism. `http://en.wikipedia.org/wiki/Task_parallelism`.