# Nizza: A Framework for Developing Real-time Streaming Multimedia Applications

Donald Tanguay, Dan Gelb, H. Harlyn Baker
Mobile and Media Systems Laboratory
HP Laboratories Palo Alto
HPL-2004-132
August 2, 2004*

media processing, video processing, dataflow architectures, multimedia middleware, streaming media

Real-time multimedia applications require processing of multiple data streams while maintaining responsiveness. Development of such applications can be greatly accelerated by the use of a middleware framework that abstracts operating system dependencies and provides optimized implementations of frequently used components.

In this paper we present the Nizza multimedia framework which enables rapid creation, analysis, and optimization of real-time media applications. Our main goal is to provide a simplified modular design without sacrificing application performance. The framework is based on the dataflow paradigm. An integrated scheduler automates parallelism among the modules and distinguishes between sequential and combinational modules in order to leverage data parallelism. Nizza measures application performance statistics, allowing rapid development of multimedia applications and identification of performance bottlenecks. Our framework is cross-platform and has been used to develop applications on the Windows, Windows Mobile, and Linux operating systems. We present several example applications that were implemented using our framework that demonstrate the performance and usability of Nizza.

* Internal Accession Date Only
Approved for External Publication

# Nizza: A Framework for Developing Real-time Streaming Multimedia Applications

Donald Tanguay, Dan Gelb, H. Harlyn Baker
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

## ABSTRACT

Real-time multimedia applications require processing of multiple data streams while maintaining responsiveness. Development of such applications can be greatly accelerated by the use of a middleware framework that abstracts operating system dependencies and provides optimized implementations of frequently used components.

In this paper we present the Nizza multimedia framework which enables rapid creation, analysis, and optimization of real-time media applications. Our main goal is to provide a simplified modular design without sacrificing application performance. The framework is based on the dataflow paradigm. An integrated scheduler automates parallelism among the modules and distinguishes between sequential and combinational modules in order to leverage data parallelism. Nizza measures application performance statistics, allowing rapid development of multimedia applications and identification of performance bottlenecks. Our framework is cross-platform and has been used to develop applications on the Windows, Windows Mobile, and Linux operating systems. We present several example applications that were implemented using our framework that demonstrate the performance and usability of Nizza.

## Keywords

Media processing, video processing, dataflow architectures, multimedia middleware, streaming media.

## 1. INTRODUCTION

Building robust systems for real-time streaming multimedia applications is difficult. A developer must overcome at least four types of challenges — *system*: isolate and manage complexity, *multimedia*: support concurrent execution on multiple data formats, *streaming*: operate on sequences of data, and *real-time*: deliver responsive performance on variable-strength platforms under varying loads. Our goal is to help the developer overcome these challenges by building applications on top of an existing middleware layer. This framework should lead to improved software design and less prototyping time without sacrificing performance.

The fundamental idea is to design application software in a dataflow style. In a dataflow design, the application is a connected graph of functional modules linked together by directed arcs. A dataflow design is well-suited for representing streaming multimedia applications: the modularity reduces complexity, the arcs represent streams of data, and the arcs can transmit multiple data formats. Clearly the first three of the above challenges are relatively easy to overcome, especially with the aid of modern object-oriented programming languages. However, the real-time requirement is much more challenging. While any application will always be responsive on an over-powered machine (*e.g.*, webcam video capture using a server-class machine), the real performance differentiators are (1) taking full advantage of multiprocessing and (2) delivering performance even when a machine is resource-limited.

Our basic approach to design and coding is to isolate the algorithms (*e.g.*, video processing or analysis) from the runtime system (*e.g.*, multithreading, synchronization). The developer concentrates on the algorithmic processing specific to the application at hand, while at the same time leveraging the framework to overcome the challenges above. In addition, such a dataflow middleware provides other software engineering benefits, like improved writability and readability (which simplifies maintenance), code reuse to leverage the work of others, better testing methodologies to simplify debugging and ensure software robustness, and increased portability to other platforms.

## 2. RELATED WORK

This work is inspired by early dynamic dataflow computers (*e.g.*, [2]) which potentially exploit the full parallelism available in a program. In such a computer, each processing node is enabled when tokens with identical tags are present at each of its inputs. Thus, process scheduling is completely determined by the availability of data. While a dataflow computer can achieve fine-grained parallelism at the instruction level, our framework operates at a much coarser granularity — that of typical multimedia samples, such as a single video frame. Unfortunately, dataflow computers are inherently unscalable because the number of nodes in the hardware limits the number of live modules in an application. Parallelism must be lost in order to implement an application with more tasks than the hardware. Similarly, because we map an arbitrarily large dataflow abstraction onto a symmetric multiprocessor (SMP) machine, the number of underlying processors limits our amount of realizable parallelism.

Signal processing software environments (such as Ptolemy [4] and Khoros [13]) have an established history of “visual dataflow pro-

gramming." A thorough review of such systems and their relationship to other dataflow styles is presented in [6]. The hierarchical structure of a DSP application can be displayed and manipulated graphically. The program can be compiled or interpreted and can be executed on an SMP machine or on specialized DSP hardware. The one-dimensional, fine-grained, deterministic nature of signal processing often allows optimal scheduling at compilation time. In our domain of generalized multimedia, the mapping of inputs to outputs may not be deterministic, and the coarseness of the samples provides no guidance for "optimal" static scheduling.

Some commercial frameworks are available for multimedia processing, including DirectShow [9], the Java Media Framework [5], and Quicktime [14]. DirectShow provides useful plug-and-play compatibility between third-party developers. For example, a commercial video conferencing application can transparently use any particular video camera if both adhere to a common DirectShow interface. Some developers are using DirectShow as a general dataflow framework, but the software is nonportable and the interface is complex because each module is a service that augments the operating system. Perhaps its biggest value is the large inventory of reusable modules. The Java Media Framework has cross-platform support, as well as integrated networking support via RTP; however, performance of heavy media (*e.g.*, video) is not competitive with other frameworks. Existing frameworks rarely provide scheduling options to the user. Because they don't have an explicit dataflow scheduler, the modules are subject to the vagaries of the OS scheduler and are often competing with each other. In a system without significant resource constraints this can be sufficient, but problems arise in constrained environments. None of these commercial offerings provide automated performance metrics.

Distributed computing extends the dataflow approach from a single machine to a network of machines. Communication between modules occurs across a network, which introduces different performance considerations. For example, bandwidth may replace computing as the limiting resource, and so different solutions are necessary. The Berkeley Continuous Media Toolkit (CMT) [8] is multi-platform and uses Tcl/TK. The open-source Network-Integrated Multimedia Middleware (NMM) project [7] has demonstrated applications with set-top boxes and wireless handhelds. Space-time Memory [12] is a high-level programming abstraction that unifies all the data distributed across a network into a single memory model that is indexed both by network location and time. This model, Stampede [11], and other distributed computing notions are merged into D-Stampede [1]. We believe a distributed computing approach can be very complementary to our framework.

Our framework has already demonstrated its utility. In [3], we built a sophisticated application that scaled with the number of users. Our dataflow graphs had 20 to 100 audio, video, and network processing modules. During tests for scalability, the two-processor desktop machines were quickly overloaded to 100% CPU utilization but maintained responsiveness throughout.

# 3. NIZZA FRAMEWORK

This section describes our design and development methodology, model of computation, implementation, and runtime performance metrics.

## 3.1 Design Methodology

Like all architectures based on the data flow paradigm, in Nizza media flows through a directed graph of computational modules. In order to view applications in this paradigm, it is important to adopt a different methodology for software development. Our suggested methodology has four steps: (1) dataflow analysis of the application to determine the signals and processing phases on those signals, (2) decomposition of the application into media representations and processing modules, (3) composition of the modules into a directed graph network, and (4) runtime management of the application graph. We describe these four steps, while highlighting how Nizza aids this process.

*Dataflow analysis.* In a streaming media application, the fundamental information content is a digital signal (*e.g.*, audio or video) that evolves over time. To perform a dataflow analysis of the application, one identifies signal sources (*e.g.*, microphone, camera, or file) and follows the transformation path each signal takes as it progresses through the application. Between each identifiable format along this transformation path, the signal undergoes a distinct phase of processing. For example, an audio signal may begin life in PCM format at the microphone source then undergo transformations into ADPCM, then UDP packets. In this example, the compression stage lies between the PCM and ADPCM formats, and the network packetization stage lies between the ADPCM and UDP formats. Analyzing each signal in this manner identifies both the signal formats and the different processing phases of the application.

*Decomposition.* Next, the application is decomposed into its constituents. The signal formats are media types, and the processing phases operate on those media types. Nizza provides two abstractions to support this decomposition: Media objects are the basic unit of data, and Task objects are the unit of processing. For each unique signal format, the developer defines a separate media type, inheriting from the Media base class such behaviors as timestamp recording, memory management, and automatic serialization. Likewise, for each novel processing phase, the developer defines a new task module that inherits the behavior of the Task base class. Inheritable behaviors include input/output buffer management and multithreaded execution and synchronization. The code inside the Task object is the algorithmic mapping from inputs to outputs and is isolated from common threading or synchronization issues, which will come for free by simply using the framework.

*Composition.* The media and tasks are now application building blocks. To build an application, the developer makes directed connections between the tasks, defining a directed graph. Each connection is a one-way transfer of a particular media type and represents a media stream. Unlike many other architectures, Nizza supports arbitrary graph topologies, including cycles. Cycles are important in any application with a feedback loop. For example, mouse motion from a display module may determine a viewpoint

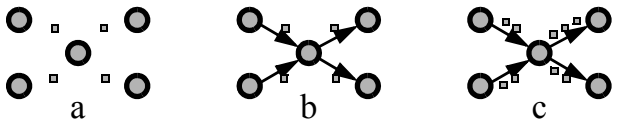



**Figure 1. Application creation. The application is (a) decomposed into its constituent tasks and signals, then built by (b) composing task dependencies into a graph structure, then (c) executed by managing the flow of signals across the task connections.**

for novel view synthesis in another module; this may in turn send a new image to the display module. In order to agree on the type of media stream, two connected Tasks may have to negotiate the media type. For example, a generalized UDP Task may accept any media type, but the video source feeding it may deliver only MPEG-4 video. Because the UDP Task is flexible, the two Tasks simply agree to send/receive the MPEG-4 video media type. In the end, the completed graph structure directly represents the task dependencies of the application.

*Graph management.* At runtime, once the tasks are connected and the media types are determined for each connection, the application is ready to execute. The program issues the start command for the application graph, and the framework's internal threads traverse the graph, performing the processing of each Task and flowing Media across the Task connections. The internal scheduler orchestrates the execution, taking advantage of parallelism when available. In addition the internal memory manager optimizes reuse of Media buffers. At a later time, the program can issue the stop command, causing the framework threads to complete execution and exit the graph. In very dynamic applications, Tasks may be added to or removed from the graph, and the start command may be reissued to continue the application with the new graph. Alternatively, the program may issue the destroy command, which recursively destroys all the Tasks in the graph.

The last three steps of this methodology are graphically depicted in Figure 1. In this example, after dataflow analysis the application is decomposed into five processing tasks (represented by dark circles) and four signal formats (represented by small light squares). The task dependencies are then made explicit during the composition step, and finally the application is executed by managing the completed task graph.

Figure 2 shows how we use Nizza to build a small application. First, the modules are instantiated, next they are connected to form a single application graph, then the graph is managed through simple graph commands, such as start, stop, and destroy.

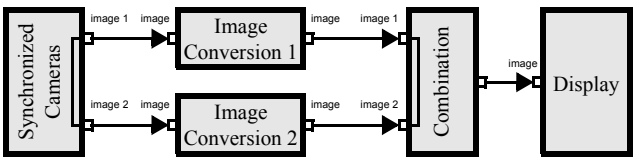



**Figure 2. Sample application. A graphical representation concisely illustrates the flow of data between modules.**

## 3.2 Model of Computation

While all dataflow architectures look very similar at the highest level of abstraction, they can behave very differently when an application has limited resources. This section describes our model of computation, which determines the application behavior in stressful situations. We have a principled approach to delivering reasonable performance on limited resources.

*Computing Service.* Nizza is a computing service by design, and this has major implications on our model of the execution environment on a single machine. First, we assume that only our framework has control of the computing resources on a machine. In other words, the framework should not be competing for CPU resources through the vagaries of an OS scheduler. Of course, in a typical non-real-time operating system, this assumption cannot possibly be true due to preemption by normal OS operations. However, by using Nizza to implement all compute-intensive applications on a particular machine, we have a reasonable approximation. Thus, we assume Nizza controls the computing resources.

In fact, our model is stronger. Since external processes should not affect Nizza performance, it is also reasonable that Nizza should not affect external processes either. This clean separation is possible by dividing the processes into two categories: computing and I/O. Computing processes take significant time and are throughput-sensitive. For example, a video codec may have a significant latency, but performance is good if it can maintain a frame-rate of 30 Hz. I/O processes, on the other hand, require less time to handle but are latency-sensitive. For example, drawing a window at a new location is relatively quick to do, but if there was a delay in performing this task, a user would notice. A similar argument applies to playing audio on an output device or capturing strokes on a keyboard. Therefore, we are careful to leave I/O operations (*e.g.*, listening to camera devices or handling window events) to the native platform. Thus, our separation of processing into computing and I/O tasks translates into two assumptions: Nizza is not competing against other compute-intensive applications, and the native platform is not competing against Nizza for I/O responsiveness.

Fortunately, an easy way to implement this model of computation is to artificially depress the priority of Nizza execution threads. This counter-intuitive approach ensures that the OS has I/O responsiveness. Since I/O is quick, the entirety of the remaining CPU time is given to Nizza, which is the only compute-intensive application. In other words, Nizza handles computation while (and only after) the OS and other standard-priority threads handle I/O.

*Execution Model.* We have established our model that Nizza has most of the computing resources on a machine. We now explain what the framework will do with those resources. In a single processor scenario, all dataflow architectures should behave in the obvious way. The CPU will work on the initial signal and propagate the signal and its descendent signals through the graph (in any valid order guided by data dependencies) until the wave of signals is entirely consumed. This procedure is repeated similarly on the next initial signal, and so on. If the average arrival rate of the new initial signals is greater than the average completion rate of each

wave, some initial signals must be dropped in order for the application to remain current (*i.e.*, to avoid continually falling behind with ever-increasing latency).

In a multiprocessor scenario, potential parallelism significantly changes the dynamic behavior of the application. In our model of a computation module, it may have internal state that is a function of previous computations. An example of this history is the tracked coordinates of a hand, where the location in the previous frame prunes the search for the location in the current frame. To allow states of arbitrary history, the code in each module must be executed sequentially (it is not thread-safe for programmer convenience). This implies that only one execution thread may be resident in a particular module at any given instant, which implies that the largest number of "live" execution threads is the number of modules in the graph. In other words, the best parallelism we can achieve in a graph of sequential modules is *task parallelism*. A computer with processors equal to the number of modules has reached the limit of usable task parallelism. Each module essentially has its own processor, and additional processors no longer improve performance. In fact, the overall application throughput is now limited by the latency of the slowest module. *Data parallelism*, on the other hand, can enjoy linear performance improvement as the number of processors increases.

Even within the limits of task parallelism, there are many options in choosing which Task to execute next. We have chosen a policy for our scheduler that favors minimal end-to-end latency. This policy is implemented by favoring descendents of the oldest initial signal.

*Combinational vs. Sequential.* Modules can be categorized by their temporal dependencies. *Combinational* modules produce output that is solely a function of the current inputs. In other words, these modules do not have any internal history of previous executions. *Sequential* modules, on the other hand, do have internal memory, and so the output may depend both on the current input and previous inputs. In this situation, the data must arrive at the inputs in the correct order. It is well-known that a sequential module can be converted to a combinational module by transferring the current state to the next execution, achieved by linking an additional output to an additional input. This conversion is useful for exposing more parallelism. However, we allow both types of modules because state transfer can be awkward or impossible for arbitrary module states (*e.g.*, an unwieldy database or the state of a device driver).

In a multiprocessor system, the choice of input processing can have a significant impact on performance. As mentioned earlier, if all modules are sequential, the best case scenario is task parallelism. In Nizza, we provide a simple mechanism for specifying combinational modules. By removing some of the sequential constraints, our scheduler can take advantage of data parallelism as well. This is particularly key for a module that is a performance bottleneck. In order to use data parallelism, the module's algorithmic code must be reentrant because multiple threads may be executing the code at the same time. Threads often vary in execution time, and so the output may not be in sequence. If the downstream module is combinational, the thread continues to run freely, taking advantage of more data parallelism. If the downstream module is sequential, however, the producing threads must block until the correct sequence is attained on the input buffer. More scalable performance, then is attained by minimizing the number and latencies of the sequential modules in an application. When possible, a large sequential module should be decomposed into a combination of a small sequential module and a large combinational module. Unfortunately, some modules can never be combinational, and their inherently sequential behavior will always limit the amount of parallelism. Such modules are typically sources (*e.g.*, a module that is triggered by an inherently-sequential input device such as a camera) or sinks (*e.g.*, an audio module that writes speech data into an output buffer).

*Push vs. Pull.* There are two major flavors of moving data through a dataflow graph: push and pull. In a push architecture, the flow is supply-driven, and data is injected into the graph at source modules (those without input pins). Either the source modules synchronously generate the initial signal themselves, or the source modules receive the initial signal asynchronously from an external trigger (*e.g.*, a camera driver). If the data comes from an asynchronous process at a rate faster than the consumption rate, flow control (*i.e.*, buffer management) must determine where to drop signals in the graph. When signals are dropped inside the graph (*i.e.*, not at the entrance of a source), the previous stages of processing on the signals wasted computing resources. On a multiprocessor, a push architecture with asynchronous sources (*e.g.*, cameras) will have better performance in general because an idle processor will be able to begin processing a new signal wave before another processor has finished processing the previous wave.

A pull architecture, on the other hand, is demand-driven. Data is requested by sinks, which need to be "rendered," and the request is propagated upstream until a module can satisfy it. Because processing only occurs when it is needed downstream, every signals is fully propagated to the end without wasting any CPU resources. However, if the sources are asynchronous and generating data faster than consumption, the straightforward pull architecture will under-utilize the CPU and throughput will be worse than the push architecture. It is also not straightforward to implement a pull architecture with a cyclic graph.

## 3.3 Implementation

Figure 3 depicts our implementation hierarchy. The SMP Abstraction Layer and the Nizza kernel form the middleware that lies between the application and operating system. The SMP Abstraction Layer insulates the kernel from the host platform, keeping the Nizza kernel platform-independent. The component library contains many generically reusable modules. The application developer has access to all levels.

*Native Platform.* At the lowest level lies the host platform, which must have three required elements: multithreading support, a timing mechanism, and an ANSI C++ compiler. Multithreading support includes a thread abstraction for controlling computation as well as the synchronization objects necessary to control the threads. Although Nizza can operate on a single processor

machine, the underlying hardware must be a symmetric multiprocessor (SMP) machine in order to benefit from parallel execution. A timing mechanism is necessary for performance analysis, such as latency measurement. Finally, an ANSI C++ compiler is necessary to generate the executables, and it must have the C++ Standard Template Library (STL, described in [10]) as we use provided abstractions (*e.g.*, string, vector, map, set, deque) throughout our framework. STL is implemented entirely of header files and is, therefore, easily ported.

*Abstraction Layer.* The SMP Abstraction Layer is the first middleware level; it simplifies porting Nizza to other platforms and operating systems. The Thread abstraction delivers the ability to name, spawn, and debug an OS thread. Actual thread creation is performed with the native platform calls, and each Thread has a log file associated with its unique name, allowing the creation of an execution trace for each thread. The Mutex and Semaphore abstractions enable synchronization of the Threads. Mutex is the standard mutual exclusion object for preventing more than one thread from simultaneous code execution, and Semaphore is a standard, efficient mechanism for signalling between Threads. The StopWatch abstraction encapsulates the ability to measure time using the platform timing functions. Measuring time is essential to performance analysis, described in Section 3.4.

*Kernel.* The second middleware level, the Nizza kernel, implements the core dataflow functionality used by all Nizza applications. It supplies the extensible Task and Media abstractions for building an application. Internally, however, the Nizza kernel also has several abstractions for managing its own complexity. First, the InputPin and OutputPin objects represent the connections between Tasks for transferring Media. Second, the Graph object manages the Tasks connected to one another and acts as the interface for graph-wide commands, such as start() and stop(). Graph commands have a single Task argument, but use connectivity to traverse the entire application graph and apply the command to each module in the graph. The MemoryManager object provides the buffers for Media objects; it tracks buffer usage, has facilities for reusing previously allocated buffers, and can report memory statistics. Finally, the Scheduler manages the execution threads that traverse the Task graphs, keeping track of computational statistics such as mean latency and throughput.

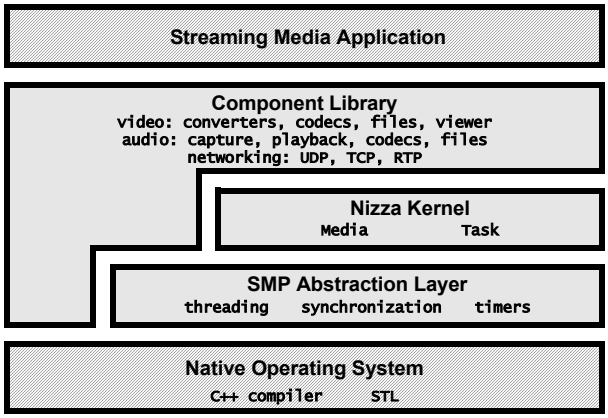



**Figure 3. Implementation hierarchy. Nizza forms the middleware between the application and native OS. While the SMP abstraction layer keeps the kernel platform-independent, some modules in the component library (*e.g.*, camera drivers) must be device-specific.**

*Component Library.* An important layer is a continually growing collection of reusable components lying between the application and kernel. Rather than reimplementing common functionality (*e.g.*, audio recording or image color-space conversion), the application developer may find useful, prebuilt Tasks from this reusable component library. Cameras, graphics, codecs, networking, etc. Leveraging the work of others is an important aspect of rapid development.

The final implementation layer is the application, which has access to all previous layers. To promote further platform-independence, the application has access to all Rock objects. In the Nizza kernel, however, only the Task and Media abstractions are accessible in order to minimize the complexity of the Nizza interface. The internal objects are accessed indirectly through Task and Media or through static Nizza procedures.

*Implementation Features.* Nizza has some distinguishing implementation features. First, there is a convenient mechanism for grouping input or output pins if it is known *a priori* that the data on those pins should always be associated together. For example, the combination module of Figure 2 will always operate on a pair of images. By placing the input pins in the same input group, the module begins operating only when both images have arrived, avoiding the need for the developer to manage and associate the input images. Because this association is known a priori, we call this *static synchronization*.

A second key implementation feature is *automatic serialization*. The Media base class has a powerful serialization procedure that can flatten any Media object, regardless of its complexity. Media can have both fixed-length fields (*e.g.*, image size, format specification) and variable-length fields (*e.g.*, image bytes, audio data). The serialization procedure is able to traverse any deep Media structure and translate it into a single flat buffer for output to a serial representation, such as a file or network stream. Likewise, the deserialization procedure can read the flattened representation and translate it back into a deep Media structure in memory.

Finally, the programmer has the ability to specify the number of execution threads in the Nizza service. In order to achieve the maximal parallelism, the number of threads should equal the number of processors. However, during debugging of an application, it is extremely helpful to use a single execution thread so that it can easily be tracked.

## 3.4 Performance Metrics

There are many different metrics for multimedia applications: startup latency, loss percentage, CPU utilization, memory usage, and throughput, just to name a few. Because we are using a dataflow architecture, we decided to focus on two metrics fundamental to pipelined systems: latency and throughput. We measure the latency per module as well as end-to-end latency. The *module latency* is the time required to execute the developer's algorithmic code in a module. To determine the relative "computing weights" of the

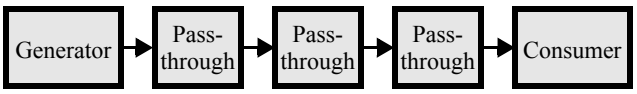



**Figure 4. Graphical representation of the trivial application.**

modules, it is useful to first run an application with a single execution thread to get an accurate measure of the natural execution times and a baseline for multiprocessor performance. The *end-to-end latency* is the time required to consume an initial signal and all of its descendants, *i.e.*, the time from the birth of the initial signal to the death of its last remaining descendant. All latencies and throughput are measured automatically by the framework.

# 4. RESULTS

We have run and analyzed three applications; all applications were run on a 700 MHz Pentium III, 6 processor machine. In the following figures, we represent sequential modules with a thick border and combinational modules with a thin border.

## 4.1 Overhead Measure

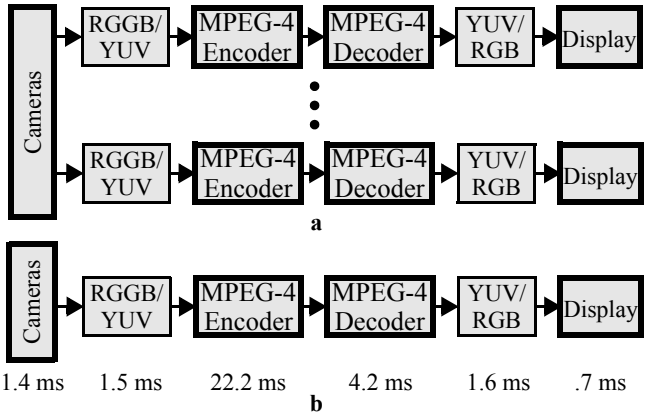
We constructed a minimal application that performs negligible computation in each module to test the overhead latencies in our system. Figure 4 illustrates the application graph. The first module in the application is a source node that generates test media consisting of a single character. The next modules are multiple processing nodes that simply push the media out to the next module and declare that they no longer need access to the media. The final module retrieves the character from the media and also declares that it is finished with the media. All modules are sequential in order to produce the worst-case latency. Runs of the trivial application on our test machine gave an average latency of less than 30 microseconds per processing node. This amount of latency overhead per module is acceptable for typical multimedia applications.

## 4.2 Video Coding

The second application is designed to simulate a multi-user/multi-stream video conferencing scenario. For this application each user is imaged by a camera and the video stream is MPEG-4 compressed, sent across a network, received at a remote client, decompressed, and displayed. Color space conversion modules are also used to convert to and from the camera color space to a YUV 4:2:0 format for MPEG-4 compression. We constructed this application quickly because the converters and codecs were available in the Nizza component library. Connecting the nodes is straightforward, and modules such as the color space converters transform input media to the specified format without the developer having to worry about issues like pixel alignment and ordering. Network serialization and deserialization is also automatic.

To enable repeatable experiments we removed the network transmission and reception from the real application to create a simplified version running on a single machine. The graph of this simplified version is shown in Figure 5a for an application using at least two synchronized cameras. To ensure a repeatable experiment the live cameras were also replaced with a stored image sequence. For the following experiments a CIF version of the standard Foreman sequence was used for each data stream. A single source is tied to each chain of modules, demonstrating a fan-out ability. To further reduce possible I/O influences the source image sequence nodes were configured to store the entire raw image sequences in memory before performance measurements. Figure 5b shows the average latencies per module for a single stream version of the video coding test application run on a single processor on the test machine.

Figure 5c shows the performance of the video coding application when multiple streams are compressed and decompressed simultaneously, one per processor. As the number of processors increases, the overall system throughput increases as well. This increase is sub-linear, since other limitations in the system such as memory and bus bandwidth are fixed. The graph shows results for enabling combinational processing nodes for this application versus all sequential modules. For the video coding application only the color conversion modules can be made combinational, as the processing intensive encode and decode modules depend on prior inputs. As a result, the speedup achieved by enabling combinational processing is limited. Our tests show that Nizza is able to provide excellent real-time performance for this application. Frame rates are sufficient for all streams, and the end-to-end



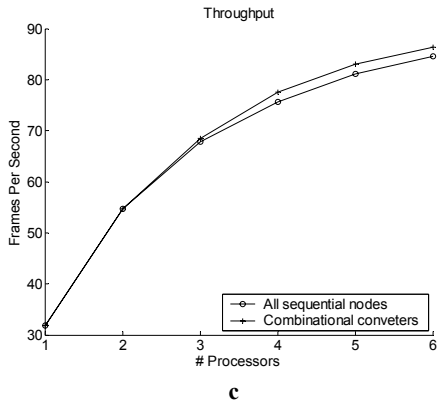




**Figure 5. Video coding: (a) Model application, with sequential modules illustrated with a thick boundary. (b) Experimental setup showing mean module latencies for a single stream. (c) Plot for number of streams equals number of processors.**

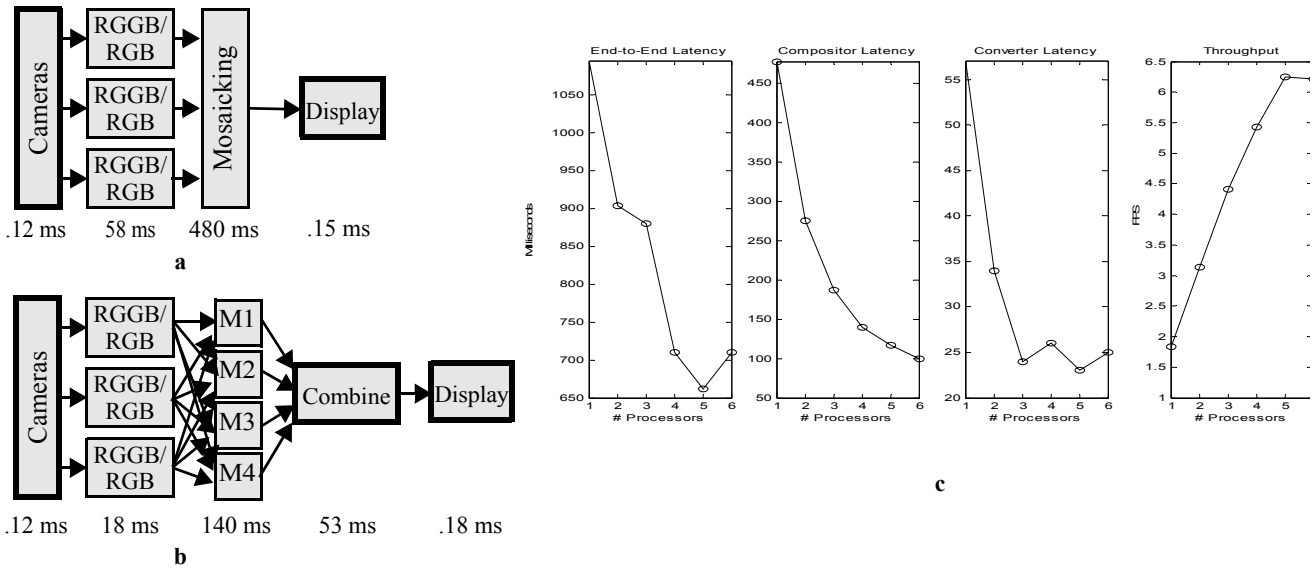



**Figure 6. Video Mosaicking: (a) model application, (b) experimental setup showing average module latencies, (c) performance plots. The sequential modules are illustrated with a thick boundary.**

latency was less than 80 milliseconds for all tests. This latency is sufficient for typical live video coding applications.

## 4.3 Panoramic Mosaicking

The third application is an example of multi-image streaming tasks where a single element of the computation dominates all others. In this case, we are constructing a panoramic mosaic from three contributing image sources. Again, these sources are stored image sequences rather than live cameras to allow repeatable performance testing. Each contributing image is passed through a converter module that maps from Bayer format to RGB. These are then input to a lookup-table-driven resampling process that composes the mosaic frame, resampling and integrating the contributing sources to a common viewpoint. The lookup table was derived offline. Figure 6a shows the structure of this task's computational graph. Mosaicking occupies over 90% of the processing time. Because only a single processor is available for this sequential task, throughput is limited to the rate of this most-demanding module. Using Nizza performance statistics, we rapidly identified the bottleneck and decided to restructure our computation. With $n$ processors available, the compositing task may be partitioned into $n$ slices, each producing $1/n^{th}$ of the resulting mosaic frame. We introduced an additional stage to combine the contributions of the separate slices. The resulting architecture is shown in Figure 6b. The times in these figures indicate the average processing time of each stage.

Figure 6c plots performance measures against number of processors (from the left): the time per produced frame, the cost of the compositing stage, the cost of the conversion stages, and the resulting framerate of the system. Notice that the end-to-end-latency and the throughput improvements stop at five processors — the point at which no additional parallelism is available. Similarly, the conversion stage, having three modules, ceases to improve once those three have separate processors for their work.

# 5. CONCLUSIONS

Nizza has proven very useful for developing real-time multimedia applications without sacrificing execution performance. Unlike other multimedia software architectures (*e.g.*, DirectShow, Java Media Framework), Nizza is designed specifically for SMP performance. A significant feature is its ability to take advantage of data parallelism, which significantly increases performance scalability over a design with only task parallelism. Second, rather than relying on the OS, the kernel has its own integrated scheduler which scheduled to minimize latency and prevents wasted CPU cycles by disallowing dropped media. Finally, a developer can easily analyze applications using the Nizza facilities for automated performance metrics.

Nizza has also demonstrated flexibility. The kernel has been ported to three platforms: WinXP, PocketPC, and Linux. It has extensible data and module abstractions and allows dynamic application building (adding/removing subgraphs). It is also easy to build new applications from reusable modules.

We would like to extend Nizza in two ways. First, the performance metrics could be fed into the scheduler to see if better performance can be achieved. Second, we would like to fold Nizza into a distributed computing architecture. We believe the configurability, library of reusable components, and the performance metrics would be valuable to a network-level service framework.

# 6. ACKNOWLEDGMENTS

We thank the contributions of the vision and graphics group, especially Irwin Sobel, who has helped us track down numerous bugs in our system. We also thank Mike Harville for testing the Linux

port of the kernel. Finally, we thank John Ankcorn for pithy discussions that lead to scheduler modifications so that we could get data parallelism.

## 7. REFERENCES

[1] Sameer Adhikari, Arnab Paul, and Umakishore Ramachandran, "D-Stampede: Distributed Programming System for Ubiquitous Computing," *22nd International Conference on Distributed Computing Systems*, July 2002.

[2] Arvind, D. Culler, R. Iannucci, V. Kathail, K. Pingali, and R. Thomas, "The tagged token dataflow architecture," Technical report, MIT Laboratory for Computer Science, 1984.

[3] Authors. Reference withheld to preserve anonymity during review process.

[4] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogeneous systems," *International Journal of Computer Simulation*, April, 1994.

[5] Java Media Framework, Sun Corporation, http://java.sun.com/products/java-media/jmf/

[6] Ed Lee and Thomas Parks, "Dataflow Process Networks," *Proceedings of the IEEE*, May 1995.

[7] Marco Lohse, Michael Repplinger, and Philipp Slusallek, "An Open Middleware Architecture for Network-Integrated Multimedia," in *Protocols and Systems for Interactive Distributed Multimedia Systems, Proceedings of IDMS/PROMS'2002 Joint International Workshops on Interactive Distributed Multimedia Systems / Protocols for Multimedia Systems*, Coimbra, Portugal, November 26th-29th, 2002

[8] Ketan Mayer-Patel and Larry Rowe, "Design and Performance of the Berkeley Continuous Media Toolkit," *Multimedia Computing and Networking 1997, Proc. SPIE 3020*, pp. 194-206

[9] Microsoft DirectShow. Microsoft Corporation, http://msdn.microsoft.com/

[10] David R. Musser, Gillmer I. Derge, and Atul Saini. *The STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*, Addison-Wesley, Boston, MA, 2001.

[11] R. Nikhil, U. Ramachandran, J. Rehg, R. Halstead, Jr., C. Joerg, and L. Kontothanassis, "Stampede: A programming system for emerging scalable interactive multimedia applications," *11th International Workshop on Languages and Compilers for Parallel Computing*, 1998.

[12] U. Ramachandran, R. S. Nikhil, N. Harel, J. M. Rehg, and K. Knobe, "Space-time memory: A parallel programming abstraction for interactive multimedia applications," In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, 1999.

[13] J. Rasure and C. Williams, "An integrated visual language and software development environment," *Journal of Visual Languages and Computing*, vol. 2, pp. 217-246, 1991.

[14] Quicktime, Apple Corporation, http://quicktime.apple.com/