Expressing and Exploiting Concurrency in Networked Applications with Aspen *

Gautam Upadhyaya, Vijay S. Pai, and Samuel P. Midkiff

Purdue University
West Lafayette, IN 47907
{gupadhya,vpai,smidkiff}@purdue.edu

Abstract

This paper presents Aspen, a high-level programming language that targets both high-productivity programming and runtime support for managing resources needed by a computation. Programs in Aspen are represented as directed graphs, where the edges are well-defined unidirectional communication channels and the nodes are instances of computational modules that process the incoming data. The resulting representation of a program closely resembles a flow chart describing the flow of computation in a server application and exposing the communication at a high level of abstraction. This strategy for program composition naturally allows parallelism and data sharing to be factored out of the core computational logic of a program, facilitating a division of labor between parallelism experts and application experts and also easing code development and maintenance. Aspen automatically and transparently supports task-level parallelism among module instances and data-level parallelism across different flows in an application or, in some cases, across different work items within a flow. Aspen automatically and adaptively allocates threads to modules according to the dynamic workload seen at those modules.

Aspen is tested using a web server and a video-on-demand (VoD) server. Both servers are compared to server applications coded in other languages (such as C, C++, and Java). The Aspen programs achieve the same functionality despite using 54–96% fewer lines of user code. Nevertheless, the Aspen version always performs competitively, with performance that is always similar to or better than previous web server implementations and that sees never more than a 10% performance degradation in the VoD server. On the other hand, some workloads see superior performance with Aspen: Aspen's runtime thread allocation strategy allows the video-on-demand server to support up to 36% more simultaneous 1 Mbps video streams than a hand-tuned C++ version.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Parallel programming

General Terms Performance, Design, Languages

*This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448, CNS-0532452, CCF-0429535, CCF-0313033, and the Purdue Research Foundation.

PPoPP'07 March 14–17, 2007, San Jose, California, USA.

Keywords Parallel programming, programming languages, network servers, resource management

1. Introduction

As cost-effective and resource-efficient PC-based servers migrate from uniprocessor architectures to multicore and multiprocessor architectures, applications must exhibit substantial concurrency to exploit the parallelism provided by the server. Unlike many application domains, explicit parallelism is a natural way to express the desired flow of computation in many network server applications. For example, web servers, database servers, web caches, and other related systems see nearly complete concurrency between the processing of independent requests. However, actually exploiting that concurrency efficiently can be difficult. Previous work has shown that using a single thread or process per simultaneous client request (as in Apache 1.x) does not scale well, as excessive context switching and thread management overheads saturate the CPU and limit performance [26]. Multiplexing a single thread across many requests in an event-driven fashion can be more efficient, but is also more difficult to program and can yield poor performance in the face of unexpected latencies such as file cache misses or page faults [26, 37]. Some strategies have used hybrids that combine event-driven execution and multithreading, but these are still limited by the difficulties of event-driven programming and the complexities of efficiently allocating resources to the various concurrent tasks [23, 26, 34, 37].

The difficulties in designing efficient software for network service applications are further compounded by limitations in existing parallelism models. A cornerstone of modern software engineering is encapsulation, yet both shared-memory and message-passing systems violate this goal. Current and proposed synchronization mechanisms in shared memory require the cooperation of all other code, while message-passing requires each process to have low-level knowledge of the data layout, implementation, and operational flow of other processes. Streaming software models avoid these problems, but they focus on computation-bound applications [21, 32]. In contrast, network service applications depend on operating system interactions with large and unpredictable latencies, requiring large and variable levels of concurrency to achieve high performance.

Modern programming languages also hinder the development and maintenance of concurrent high-performance network service applications. Interactions with the system such as network operations, thread creation, inter-thread communication, and resource allocation are handled by library code rather than language primitives. Consequently, the programmer must explicitly manage numerous cumbersome details, such as binding threads to subtasks, coordinating data movement, opening different kinds of network

connections, and load-balancing. Further, these library calls are also intermingled with complex code that performs error handling, scheduling and event handling, resource allocation, and the core application logic. By obscuring the overall flow and intent of the program, current programming languages make it difficult both for the compiler to automate and optimize the actual process of execution and for the programmer to maintain code written by others or not recently used. Such a design strategy is error-prone, and errors can lead to poor performance, resource leaks, and applications that are more prone to security violations.

This paper presents Aspen, a high-level programming language that targets both high-productivity programming and runtime support for managing resources needed by the computation. Aspen eschews shared memory in favor of a message-based interface, but uses a highly abstract form of message passing that makes its semantics clear to the Aspen programmer, compiler and runtime system. Aspen requires the parallel structure of a program to be specified independently of the computational logic of the program. Specifically, Aspen allows *communication graphs* for an application to be specified, where edges in the graph are well defined *communication channels* and nodes are instances of *modules* that specify computation to be performed on the communicated data. A communication graph is akin to a flow chart describing the overall flow of computation in a server application, and allows the communication to be cleanly specified at a high level of abstraction. Aspen automatically and transparently supports data-level parallelism across different flows in an application, and across different work items, or elements, within a flow. Just as importantly, Aspen allows the programmer of a function to be oblivious to the actual communication mechanisms and the global communication structure, and it allows the compiler to have sufficient information to generate efficient communication and calls into the Aspen runtime to allow efficient management of resources. Aspen automatically and adaptively allocates threads to modules according to the dynamic workload seen at those modules.

Aspen is tested using a web server and a video-on-demand (VoD) server. The web server tests have two components, which test Aspen's ability to serve static and dynamic requests. To test the web server's ability to serve dynamic requests, Aspen is compared to Apache-2 using a workload based on the "support" portion of the SPECweb2005 HTTP performance benchmark. The static tests employ a subset of the SPECweb99 workload to compare Aspen to the Flash and Haboob web servers and to a web server constructed using the Flux programming language. The VoD server is compared to a hand-tuned C++ server. The Aspen programs achieve the same functionality as the other servers, despite using 54–96% fewer lines of user-written code. Nevertheless, the Aspen servers always perform similar to or better than the other servers for the web workloads and never see more than a 10% performance degradation over the hand-tuned server for the VoD application. On the other hand, some workloads see superior performance with Aspen: Aspen's runtime thread allocation strategy allows the video-on-demand server to support 36% more simultaneous 1 Mbps video streams than the hand-tuned C++ version and, at maximum load, to serve 9% more bandwidth than the next application for the static web server experiments.

This paper makes the following contributions.

- It describes the Aspen language, and shows how Aspen's high level abstractions enable dynamic management of load-based thread allocation and dynamic management of other resources such as sockets;
- It describes the different forms of parallelism supported by Aspen, and shows how Aspen's high level abstractions aid in the discovery of that parallelism, and explains how the Aspen runtime aids in the efficient exploitation of that parallelism;
- It describes the Aspen runtime, explaining how much of the complexity of managing system resources can be removed from the programmer and placed onto the runtime system to enable adaptive thread allocation;
- It provides experimental results showing that Aspen achieves high performance, compact code, and code that is easy to augment with new functionality.

2. Forms of Concurrency in Networked Applications

Networked applications have various types of concurrency. For example, typical network servers consist of various tasks, and the processing of different requests can be pipelined through these tasks. Additionally, requests belonging to different TCP connections (or other sorts of IP flows) may typically be satisfied in parallel. In some situations, individual work elements coming from a single connection may also be performed in parallel. The Aspen programming language is designed to support each of these forms of concurrency, which are termed *task-level*, *flow-level*, and *element-level* parallelism, respectively.

Aspen programs are composed of a sequence of operations, either built-in or user defined. Each operation is defined as a *module*, which is similar (but not identical) to a class in an object oriented language. Each module defines a `data` section, an `init` function, and a `run` function. Aspen programmers connect the different modules into a task graph, as shown in Figure 1. Each occurrence of a module in the task graph is a *module instance*. Module instances are connected by queues, with each module having a single input queue, and one or more named output queues. The Aspen programming model treats each Aspen module instance as having a different memory space from all other Aspen module instances – the data specified with the module exists independently in each module instance. Thus, Aspen supports a distributed memory programming model at the level of module instances. Work elements arrive at the module instance on its input queue, triggering an invocation of the module's `run` function (sometimes referred to as an invocation of the module). The `run` function reads zero or more items from the input queue, performs its computation, and then writes zero or more items to any of the module's output queues.

By specifying the task graph, the Aspen programmer explicitly specifies task-level parallelism in the Aspen program. The data section can be used to store state in variables for a given module instance. Any inter-instance data sharing must be explicitly performed by sending messages on the output queues, allowing for straightforward pipelining of request processing through connected module instances and for concurrent execution of module instances that are not connected. Aspen's distributed memory programming model makes this task parallelism trivial to exploit since no synchronization is necessary between different module instances. It is also natural to have programmers express task parallelism explicitly since the programmer is already responsible for writing the code for each task: explicit support for task parallelism through a task graph more directly resembles a programmer's own flowchart than a convoluted sequence of function calls. Task parallelism could be thwarted by sharing through files, which is harder to detect. Conflicting file operations, i.e. having a file opened for write by more than one instance, and having a file open for writing and reading by different instances, is prohibited in Aspen. Because file operations are relatively heavyweight, however, monitoring at runtime the location of files that are opened will incur little overhead. When it is discovered that a module instance is about to open a file for reading (writing) that is already open for writing (reading or

writing) by another module that is executing with task parallelism, an error can be issued. We note that, for example, a program could use a single module instance to actually update and read from a file, distributing the results of its reads to other modules and receiving write requests from those other modules.

Aspen's primary target applications are network service codes, which usually operate on either TCP connections (or other ordered flows) or unordered IP packets. The former includes examples such as web server codes; the latter includes port-based firewalls. Systems such as network intrusion detection combine both categories. In the first category, work elements from the same flow must be processed in-order, but there are no dependences between work elements from different flows. Applications in the second category see no dependences at all between work elements. These types of concurrency exemplify the flow-level and element-level parallelism described above. A unique feature of Aspen is explicit support for intra-flow sharing. If a variable is declared in the data section of the module declaration with the keyword `per-flow`, then a separate copy of this variable is automatically created for each data flow that accesses this variable. (The implementation of this feature is described in Section 3.) Aspen can exploit flow-level or element-level parallelism under certain conditions, described below, based on the state visible to an Aspen program.

The state of an Aspen program at some time t is composed of three parts: the queues in the system, the variables in each module instance, and any files or network connections that are open at time t. Files or connections that were open prior to time t, but that are no longer open, are no longer part of the program state (but may, of course, have affected the state of variables, queues, and other open files or network conditions). Automatic stack-allocated variables are alive during the invocation of their creating function, per-flow variables are alive while a given network flow exists, and the instance-wide data variables are alive throughout the execution.

Data parallelism across flows can be utilized by Aspen within a module instance when operations on each flow are independent. More concretely, independent flows cannot share file pointers if one flow may write to the file. As well, network connections, per-flow variables, or instance variables cannot be shared. We note that flows share per-flow variables by specifying another flow's identifier when accessing the variable, as described in Section 4. Thus, when the computation on one flow cannot affect the computation on another flow and when a given module instance has no instance-wide data sharing, flow-level data parallelism can be exploited. Aspen expresses this parallelism by allocating multiple threads to a module instance, with the computation of one or more flows bound to each thread at any given time. Binding flows to threads preserves sequential processing of the data of a flow.

Parallelism between flows in a module instance is inhibited only if the flows share state within the instance. This can occur in four ways: (i) two invocations of the module instance processing different flows write to the same shared instance variable; (ii) an invocation of the module instance affects the per-flow variables of a different flow than the flow of some element dequeued in that invocation; (iii) state is shared through files; and (iv) there are network accesses. The first can be conservatively checked by determining if there are any writes to shared instance variables. These are known statically (by declaration), and if this is not done, (i) cannot be a reason to inhibit parallelism. If there are such writes, the system may be able to determine if values to the variable are always assigned within an invocation of a module instance before use. If not, then flow parallelism will not be exploited for this module instance. The second condition can be checked by determining if, for every element dequeued in this invocation of the module instance, the key to a read or write of a per-flow variable is always the flow identifier. This can be done by a form of constant propagation, where the flow key value on an input queue item is considered a constant by the analysis if unchanged. If more than one item is dequeued, then the flow key value can conservatively be assumed to be non-constant. Having the same file opened for both reading and writing within a given module instance will disable flow parallelism. This can be checked more accurately by using a combination of compile time dataflow analysis and runtime checking. Specifically, at compile time the flow of file names appearing in open statements, and the file handles of opened files that cross modules (via queuing and dequeuing) and into per-flow data values can be tracked via a live value dataflow analysis. If at any time during the program execution the same file name exists in a global module instance variable for more than one flow, data parallelism across flows can not be utilized. Further, if more than one flow can access a file name used in an open, or a file handle, via a dequeue operation, data parallelism across flows cannot be exploited. These two conditions can be checked by monitoring, at runtime, the enqueue/dequeue operations and global variable values identified by the dataflow analysis. Finally, a direct network connection (but not a flow between a module and a built-in Aspen network interface module, discussed later) will disable flow parallelism.

Data parallelism within a flow can also be exploited by an Aspen program. Intuitively, this element-level parallelism can be exploited within a module instance when invocations of the instance are stateless, i.e. when no state persists across invocations of the instance. This is true when no files, network connections, per-flow variable values, or instance-wide variable values are shared between two invocations of a module instance. This can be conservatively checked by noting the absence of file I/O operations in the module and of assignments to or uses of instance or per-flow data variables. When element parallelism is being exploited, the module instance operates in an almost side effect free manner; the only side effects are elements removed from the input queue and placed on the output queue. File and network access is handled as with flow-level parallelism.

The lack of shared memory between module instances makes the detection of parallelism in the absence of file I/O a purely local problem. This has two important effects. First, because inter-module whole program analyses are not necessary, data flow information is likely to be more accurate, and the parallelism detection is more likely to be successful. Second, the programmer is restrained from coding the problem in a way that leads to dependences and interference across module instances. Additionally, the use of per-flow variables encourages the programmer to segregate the state between flows, and allows the compiler to detect parallelism by tracking the values of variables used as keys.

3. Design of the Aspen Language

Aspen was designed to allow programmers to specify concurrency among tasks and data flows by allowing components to be composed structurally with explicit communication queues representing the passing of information in the style of task flowcharts. This allows system designers to program in the same way they would draw the system on a whiteboard. Just as importantly, Aspen allows the description of parallelism to be separated from the core sequential computational logic that constitutes the nodes (module instances) of the workflow graph. In Aspen, the programmer is not required to explicitly manage the resources associated with concurrency or inter-node communication. The actual management of these resources is handled by the compiler-generated and library code.

There is no sharing of data in Aspen across module instances. This allows Aspen programmers to write modules whose data is encapsulated against changes caused by code in other modules. This

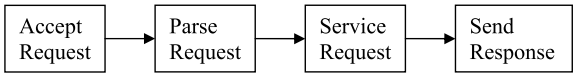

Figure 1. Flowchart for static-content Web Server

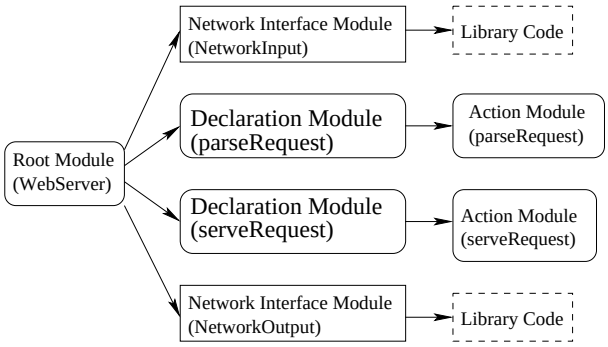

Figure 2. Static structure of an Aspen program, showing the organization of different types of code modules

facilitates code reuse and encourages a compositional approach to constructing parallel applications.

Application Structure. Aspen programs consist of modules connected through queues. Figure 1 gives a flowchart for the processing of a request in a static-content Web server. Such diagrams are a conceptually easy way of describing concurrency in a network service, and concurrency within and across requests or tasks enables a network service to achieve good throughput and utilization of the host machine.

An Aspen program is hierarchical, with a *root* defining the interaction between the various modules present in the system. Module declarations are separated from their definitions and contain various constraints that the programmer deems appropriate to the module. The module definition contains the procedures that make up the functionality of the module including the `init` procedure that initializes the module, the `run` procedure that contains the code to be executed as work flows through the system, and the `data` section which specifies the variables associated with the module instance and with specific flows passing through the module instance. Other procedures may also be defined by the programmer. The static structure of an Aspen application is shown in Figure 2.

One of the most valuable features in Aspen for programmers targeting network service applications is language-level support for per-flow data variables in a given module instance. Rather than having to maintain explicit arrays or hash tables of per-flow data, the programmer only needs to declare a single variable as being per-flow and the compiler automatically privatizes the variable for each flow. The current implementation converts accesses to per-flow variables into accesses to a hash table indexed by flow identifier. For each module instance, all per-flow variables for a single flow are stored as a `struct` in a hash table indexed by the flow identifier. Thus, one hash table reference per module instance invocation can retrieve all the variables in the structure for a given flow, allowing accesses to these variables to incur only a very low overhead (shown in Section 6 to be approximately 1%).

An additional feature aimed at network applications is primitive, language-level support for network communication. This support is provided with primitive modules called `NetworkInput` and `NetworkOutput` that represent communication into and out of a server application through a specific socket. These modules have special commands to accept and initiate connections, to

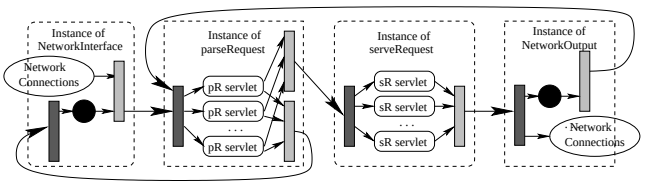

Figure 3. The dynamic structure of an Aspen program showing instances of modules, servlets, and their interaction via queues.

```
1. Module WebServer requires
2. Module NetworkInput, parseRequest,
3. serveRequest, NetworkOutput {
4.   void initialize( ) {
5.     NetworkInput nI(Input is tcp:9877);
6.     parseRequest pR;
7.     serveRequest sR;
8.     NetworkOutput nO;
9.     flow: nI ||| pR ||| sR ||| nO;
10.    flow: pR.altQueue ||| nI;
11.    flow: nO.feed ||| pR;
     }
    }
```

Figure 4. Sample application code for an Aspen root module. This acts as a communication specifier between module instances.

declare interest in receiving data from a connection, and to actually send data to a connection. The `NetworkInput` module passes out newly accepted connections and received data, while the `NetworkOutput` module has a feedback output queue to indicate the completion of a transmission.

Aspen runtime support. The Aspen runtime is responsible for ensuring messages are passed efficiently and quickly within the system. It does so by using explicit message queues declared by the programmer. Synchronized access to these queues is handled by the runtime; the programmer merely states intentions of enqueuing or dequeuing a message.

Figure 3 shows the dynamic structure of an Aspen program. Dynamically, the actions taken by module instances in Aspen are executed by *servlets* — independent threads that share the same communication sources and sinks. For module instances that support flow-level or element-level parallelism, the Aspen runtime is responsible for adaptively allocating servlet threads based on the load seen by each instance. Aspen's adaptive thread allocation will dynamically tune the number of threads for each multithreaded module by spawning or killing threads as necessary to satisfy varying load levels. This thread allocation strategy makes use of a number of equations to make decisions quickly and efficiently and includes built-in precautions against extreme transient responses. For modules with flow-level parallelism, the Aspen runtime is also responsible for insuring that multiple work elements from the same flow are steered to the same servlet so that they will be processed in-order. If a servlet completes the processing of all presently seen work elements from a flow, that flow can be reassigned to a different servlet once the next work element arrives. Flows must also be reassigned under certain conditions when servlets are spawned or killed; the details of this process are described in Section 4.

All socket descriptors are automatically garbage-collected in Aspen. Aspen uses reference-counted *smart pointers* to encapsulate instances of such descriptors, closing them when no more references to them exist anywhere in the Aspen system. This frees the programmer from having to worry about when it is safe to close a descriptor.

```
1. Module parseRequest {
2.   void initialize() {
3.     ... initialization code
4.     return m1;
     }
5.   void run() {
6.     QueueElement q1;
       ... other declarations
7.     dequeue() >>> q1;
8.     switch(GET_TYPE(q1)) {
9.     case input_sock:
          ...
10.      QueueElement(FILE(*m2)) >>> q1;
11.      send q1;
12.      break;
         ...
       }
     }
   }
```

Figure 5. Sample action code for a web server module in Aspen.

Coding considerations. Figure 4 shows how the Aspen `Webserver` root module is constructed. This root module sets up the flow graph depicted in Figure 3. This module is only a communication specifier; it contains no executable statements. Lines 1–3 declare the name of the module and list the modules that it depends on — the names of the modules that are composed to form the Aspen program. The internal workings of the required modules remain hidden; only their interface queues are exposed. Lines 5–8 instantiate modules and bind them to identifiers, while lines 9–11 declare the flow of work between module instances. Each module instance has associated with it a single input and one or more output queues through which all inter-module communication occurs. The Aspen pipe operator ("|||") specifies the connections between the default or named output queues of one module instance and the input queue of another. The programmer need only specify a handful of lines in the root module to instantiate modules and bind the instances together; the actual data transfers are managed by the runtime.

Because Aspen resources are largely managed by the runtime, the resource specifiers for a module are quite simple. In particular, a resource specifier can declare two types of resources: queues and data elements (i.e., instance-wide and per-flow data variables).

Since modules may have multiple output queues, any output queues beyond the default must be explicitly declared. Thus, the text:

```
Declare Module parseRequest {
   Output:
      Output Queue is altQueue;
```

declares an output queue named `altQueue` associated with the `parseRequest` module. Data elements are declared using essentially a C/C++ notation. Special directives are also used to specify which type of parallelism (task-level, flow-level, or element-level) is supported by a module.

Executable code in modules. Figure 5 shows an action specifier for an Aspen module. Actions taken by Aspen modules are coded in C++ with Aspen extensions. Every module includes an `init` function that is analogous to a constructor for each module instance. The `run` function is invoked whenever a new piece of work arrives on the module's input queue. Lines 7 and 10 show a message being placed in a `QueueElement` data structure. Line 11 sends this structure on the default output queue; the programmer would have used `send q1 on` *name* to send the queue element on a named output queue. Aspen also provides a sending mechanism called `send and discard` that informs the runtime that the data referenced by the queue element will no longer be used in this module invocation after the message is enqueued. In all of these cases, the module programmer does not need to know the functionality of the other module connected to the output queue or the topology of data flow among modules. Thus, module programmers are insulated from the overall system design. This design satisfies Aspen's goal of separating the specification of concurrency from the core application logic.

4. Prototype Implementation

The Aspen prototype compiler is a multi-pass source-to-source translator written with ANTLR/JAVA. The input to the compiler is a source file composed of the modules used in the program while the output is the generated standard C++ code to implement the application. Although compiler analysis could automatically discover whether a given module has task-level, flow-level, or data element-level parallelism by checking the conditions described in Section 2, the prototype currently requires the programmer to specify that explicitly.

Module implementation. Modules in Aspen are implemented as C++ classes. The constructor invokes the `init` function provided by the programmer, and the `run` function is invoked by a system-level controller. Instance-wide variables specified in the data section of the module declaration become ordinary class variables. Accesses to per-flow variables, however, are converted to accesses to data stored in a common centralized hash table. The Aspen compiler automatically adds a hash table lookup/insert operation into the `run` function of a module with per-flow data variables. The hash table is indexed with the flow identifier, and the stored data is a structure containing all the per-flow state.

Primitive modules for networking. The `NetworkInput` module uses the UNIX `select` call to check socket descriptors for new data or connection establishment requests. It also accepts messages from modules instructing it when to add to or remove from the `select` descriptor sets. Upon accepting a new client, it adds the (newly created) socket descriptor to the list of descriptors it is selecting on. If it receives data on an existing socket, it encapsulates the data in a message and sends the message downstream to the next module instance. The `NetworkOutput` module correspondingly receives messages specifying data to be sent on specific socket descriptors. The messages can either contain actual data or a file descriptor and offset for use with the Unix `sendfile` call.

Inter-module communication. Aspen implements communication between module instances (e.g. "|||") using queues. To enforce the non-shared memory semantics of Aspen, a datum to be enqueued is copied, and the copy is placed on the queue. The current Aspen system is running on a shared memory system, so a pointer to the copied data, rather than another copy, can be enqueued. In the case of `send and discard`, Aspen does not even need to make a copy; it can enqueue a pointer to the original data since that data will no longer be used by the sending module instance. Because the queues are part of the Aspen system code installed on a machine, queuing and dequeuing on a distributed memory machine can be implemented transparently to the programmer and to the generated C++ code. The queues themselves are extensions of the `deque` structure provided by the C++ Standard Template Library (STL). To allow for multithreaded and multiprocessor execution, the queue manipulation functions are internally protected using Pthreads locks and condition variables.

Adaptive thread allocation. The system responds to an increasing load on some module instance by creating additional threads (servlets) to perform the work of the module instance, and subsequently kills threads (by sending them to a per-module thread pool) when the load reduces and they are no longer needed. This

adaptive thread allocation is done in a controlled fashion so that large numbers of threads are not being continuously created and killed. The system computes a desirable range for the number of threads using the equations below, spawning or killing threads as needed to remain in the range. To reduce the overhead associated with this monitoring, a thread spawn/kill decision, and the associated computation, is only performed every k enqueues (10 in the current system).

Inputs to the equation for computing the upper and lower bounds for a range are:

- R: the arrival *rate* of work to the module instance's queue, averaged over the last k enqueues.
- S: the average service time per work item for the module instance, averaged over the last 100 work items.
- $LOW, HIGH$: values that scale the range and therefore affect the stability of the number of threads. As the difference between low and high increases, the range of acceptable thread numbers increases, and the number of threads created and destroyed goes down. These values are set to 1.0 and 20.0, respectively, in the system.
- ν_{spawn}, ν_{kill} are the global *spawn viscosity* and *kill viscosity*. These provide the system with a mechanism to control the rate of spawning or destroying threads by introducing a "drag" or "viscosity" to those changes. These parameters influence the system by temporarily increasing the acceptable range of threads.

The range lower and upper bounds (*L* and *U*) are given by:

$$\begin{aligned} L &= \frac{R \cdot S}{LOW \cdot \nu_{spawn}} \\ U &= (R \cdot S) \times (HIGH \cdot \nu_{kill}) \end{aligned}$$

Intuitively, as the amount of work being done by an instance increases (or decreases) ($R \cdot S$), the number of threads allocated to that instance increases (or decreases), damped by the system-wide viscosity factor. Every 100 ms a *Monitor* thread awakens and computes the ν_{spawn} and ν_{kill} factors using the following equations:

$$\begin{aligned} \Delta &= |\,(totalThreads - averageThreads)\,| \\ \rho &= \mathrm{sgn}(totalThreads - averageThreads) \\ \nu_{spawn} &= e^{\Delta^{\rho}} \\ \nu_{kill} &= e^{\Delta^{-\rho}} \end{aligned}$$

where $totalThreads$ and $averageThreads$ indicate the current number of threads and average number of threads present in the system, respectively. More detailed schedulers are a matter of further research but could follow policies used in other contexts, such as the IBM OS/390 Workload Manager [10].

Thread Pooling. When the scheduler determines that a thread is no longer needed, it sends the thread to a (per-module) "thread pool" instead of allowing it to die. Future requests for thread allocation are first satisfied, if possible, from the per-module thread pool. New threads are spawned only when the thread pool is empty. This approach allows us to lower the cost of spawning or killing a thread.

Additional features. All socket descriptors and strings are automatically garbage-collected in Aspen. Aspen uses *smart pointers* to encapsulate instances of such descriptors and strings and disallow manipulation of the "naked" descriptor (or string). This approach encounters some additional overhead in going through the smart pointer whenever the referenced data is accessed. The compensating benefit is that now the programmer (and indeed, the Aspen system itself) is freed from the responsibility of knowing when it is safe to close a descriptor or delete a string; such actions will be performed automatically when the last reference to the smart pointer is deleted.

5. Experimental Methodology

This section discusses the methodology used to test the Aspen prototype implementation and compares it to existing design methodologies. The implementation is evaluated using a web server and a video-on-demand (VoD) server.

5.1 Web Server

The web server portion of the evaluation consists of two separate benchmarks. In the first, the workload is the "support" portion of the SPECweb2005 HTTP performance benchmark, which includes both static and dynamic content [30]. In the second, the workload is based on the static portion of the SPECweb99 HTTP performance benchmark [29]. Both workloads are described in greater detail below.

5.1.1 SPECweb2005

Aspen is tested first by evaluating the performance of a web server, with three different codes under test:

- the freely-available Apache 2.0 web server, which uses a combination of events, threads and processes to achieve high concurrency and low overhead, in conjunction with the "mod_php" Apache module to service PHP requests.
- the Apache 2.0 web server, in conjunction with an external PHP server to service PHP requests. The servers communicate using the FastCGI interface.
- a web server written in Aspen, in conjunction with an external PHP server to service PHP requests. The servers communicate using the FastCGI interface.

We use the "support" portion of the SPECweb2005 HTTP performance benchmark. To mimic the behavior of real web sites, SPECweb uses a Zipf-distribution URL popularity model; the frequency of access to the document of rank r is proportional to $r^{-\alpha}$, where α is termed the Zipf parameter. The default Zipf parameter for SPECweb 2005 is 1.2, but we also test parameter values of 0.4 (less locality, broader working set) and 2.0 (more locality, smaller working set). The client workload has numerous features to represent real web-browser behavior, including multiple simultaneous user sessions, persistent connections, limited line speeds, and inter-request think times. The test has 3 peak phases separated by idle periods, with ramp-up and ramp-down between a peak phase and an idle period. A more thorough discussion of the design of the benchmark and of the "support" portion of the test appears in the SPECweb2005 design document [30].

5.1.2 SPECweb99

The web server is also tested using an internal benchmark that is similar to the static portion of the SPECweb99 HTTP performance benchmark [29]; similar benchmarks have also been used by other projects [6, 22, 37]. The Aspen web server is compared against three other web servers:

- the Flash web server [26]
- a web server developed using the Flux programming language [6]
- the Haboob web server [37]

Flash uses an asymmetric event-driven architecture (AMPED) to multiplex between different connections. It makes uses of the `mmap` UNIX system call to cache file contents and lazily `munmap`'s them in a least-recently-used fashion. The web server developed using

the Flux language is a multithreaded web server using a fixed pool of worker threads to multiplex between HTTP requests. The Flux runtime allows event-driven, batched, and single-thread-per-connection web servers to be developed as well, but we experienced the best performance using a thread pool of size 50. The Haboob web server is built on the SEDA software architecture [37]. SEDA allows for the inclusion of resource controllers that allocate threads to the various stages. These resource controllers must be specified by the programmer explicitly, in contrast to Aspen's adaptive system. For this paper, we have used the default controllers provided with the Haboob distribution. Flash, Flux, and Haboob were not considered for the SPECweb2005-based workload because they do not directly support FastCGI-based dynamic content generation. It should be noted that of the four web servers being tested, Flash and Haboob employ application-level caching to various degrees while Flux and Aspen do not. The clients request web pages using a distribution specified by the SPECweb99 suite. Each client closes the connection after 5 request-response pairs and opens a new connection before continuing. The total size of the working-set is approximately 3.3 GB, and files are requested according to a Zipf popularity distribution with a Zipf parameter of 1.0. Due to the probabilistic nature of the SPECweb99 workload, not all files are likely to be requested within a given period of time. Thus, even though the entire working-set may not fit in memory, application-level caching is still effective because some pages are accessed far more often than others.

5.2 Video-On-Demand

Aspen is also tested using a video-on-demand (VoD) server that aims to deliver the maximum possible number of simultaneous independent streams at a specified bitrate. This workload is very disk-intensive. The VoD server is tested with target bitrates of 1 Mbps (iPod quality), 6 Mbps (DVD-quality), 12 Mbps, and 18 Mbps. The performance of the Aspen code is compared against a C++ implementation that uses a single thread per connection to tolerate disk latency. Unlike the web server, the VoD server has a quality-of-service constraint. In our tests, the server sends roughly 1 minute of content as an initial buffering phase; after that point, it repeatedly sends out chunks consisting of 5 seconds of data at the target bitrate. The goal is for data delivery to remain ahead of the client's data consumption so that the client always has data ready to play. The performance results report the maximum possible number of simultaneous connections that can be supported without ever allowing any client connection to run out of buffered data.

Both server applications run on a SunFire x4100 server with two 2.2 Ghz dual-core AMD Opteron processors (four processors total). The operating system is Linux kernel version 2.6.8-dol-03-11-amd64-k8-smp as distributed by Debian for the x86-64 platform. The system includes 4 GB of DRAM and four Gigabit Ethernet network interfaces. The web server is always tested in a 4-disk configuration, while the VoD server is tested with 1, 4, and 8 disks since performance is highly disk-dependent. The SPECweb and VoD clients run on up to 3 independent PCs connected over a Gigabit Ethernet switch to the server; the clients are never allowed to be the bottleneck in our tests.

6. Experimental Results

This section discusses the results of experimentation with Aspen using the web and VoD servers.

6.1 Web Server Performance

6.1.1 SPECweb2005

Effectiveness of the Aspen runtime. Figure 6 shows how the Aspen adaptive thread allocation policy responds to the SPECweb2005-based workload on the web server. The X axis represents time elapsed since the start of the run. This chart plots the instantaneous bandwidth of data delivered by the server, the instantaneous number of threads used in the server, and the average number of threads. The Y-axis for the thread curves represents the number of threads, while the Y-axis for the bandwidth curve represents the bandwidth in Mbps. The peak bandwidth during the experiment nears 3 Gbps at times. After the initial phase during which the scheduler "learns" the prevalent system conditions, the number of threads used by Aspen adjusts to support the sustained bandwidth needs of the peak phases and to drop off between peaks. As bandwidth ramps up between an idle phase and a peak phase, Aspen responds within seconds by creating new threads (possibly by allocating them from the per-module thread pools) to service the increased workload. Nevertheless, the viscosity in the adaptive thread allocation scheme allows Aspen to avoid spawning and killing with every bandwidth variation, thus reducing the overhead of thread management.

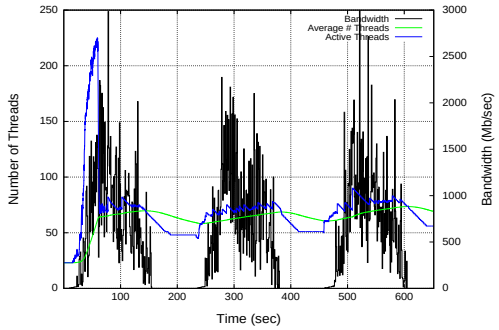

Figure 6. Effectiveness of adaptive thread allocation policy in Web server under default SPECweb-based workload

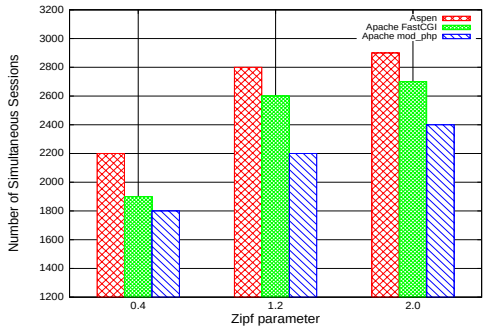

Figure 7. Performance comparison of web servers for various Zipf parameters for the support section of SPECweb2005. A greater Zipf parameter indicates more locality and a smaller working set; the default SPECweb2005 parameter is 1.2

Performance comparison. Figure 7 compares the maximum number of simultaneous client sessions achieved by the Apache-2 server with mod_php, the Apache-2 server with FastCGI and the Aspen server with FastCGI during the peak phase of the SPECweb2005 Support workload. We report the number of simultaneous client sessions (instead of an alternate metric such as latency) because that is the standard SPECweb2005 metric. The default SPECweb test uses a URL popularity distribution with a

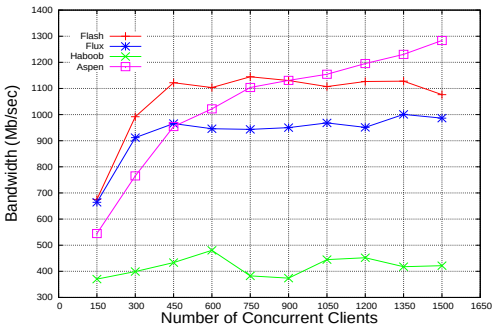

Figure 8. Performance comparison (bandwidth) of various web servers for the static SPECweb99-like benchmark

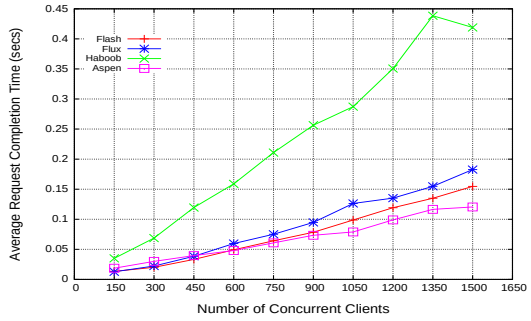

Figure 9. Performance comparison (average request completion time) of various web servers for the static SPECweb99-like benchmark

Zipf parameter of 1.2, but the tests reported here also include Zipf parameters of 0.4 and 2.0. The X-axis shows the Zipf parameter, while the Y-axis gives the number of simultaneous sessions. There are 3 bars for each Zipf parameter value, corresponding to Aspen, Apache-2 with FastCGI and Apache-2 with mod_php. As expected, greater Zipf parameters allow greater locality and thus greater performance. Aspen consistently outperforms Apache-2. It must be noted that, at the highest values of simultaneous sessions, performance is limited (in the FastCGI case for both Aspen and Apache-2) by the PHP processes, which consume almost all of the CPU resources. Even so, Aspen's adaptive thread allocation strategy allows it to outperform Apache-2 with FastCGI because it carefully and adaptively selects the number of threads allocated to the web server, allowing the maximum amount of remaining CPU resources to be devoted to the PHP processes.

6.1.2 SPECweb99

Figure 8 compares the bandwidth served by the Flash, Haboob, Flux and Aspen web servers for the static SPECweb99-like benchmark. The X-axis depicts the number of simultaneous clients used. The Y-axis depicts the bandwidth served, in Mb/sec. As can be seen, the Aspen web server is highly scalable, with performance continuing to improve over the entire range of client numbers. In contrast, while the Flash and Flux web servers ramp up faster than Aspen, their performance levels peak before reaching a larger number of simultaneous clients. Haboob's performance is substantially lower for all test cases, confirming earlier results by others [6].

Figure 9 compares the average time elapsed between the sending of a request and the reading of the response in its entirety for each of the Flash, Haboob, Flux and Aspen web servers for the static SPECweb99-like benchmark. The X-axis depicts the number of simultaneous clients used while the Y-axis depicts the average request completion time, in seconds. As shown, Aspen's performance is competitive with or slightly better than the other web servers.

6.2 VOD Server Performance

Figure 10 shows the performance of the VoD server for a target bitrate of 1 Mbps. The numbers on the X axis represent the number of disks used in the configuration (1–8), while the numbers on the Y axis represent the number of simultaneous client connections that can be supported successfully. Each system configuration has a bar for the Aspen version and a separate one for the hand-coded C++ version. The system with 1 disk sees similar performance for both the Aspen and hand-coded versions. However, Aspen outperforms the hand-coded version by 14% for the system with 4 disks and by 36% with 8 disks. This is because the operating system scheduler cannot efficiently support the number of threads needed in the hand-coded version and the CPU becomes saturated; recall that this version uses a thread per connection to tolerate disk latencies. The Aspen version uses far fewer threads because it only creates new threads when needed. Thus, even though all connections still go to disk, the requests can be time-multiplexed onto a smaller number of threads.

Figure 11 compares the performance of the Aspen and hand-coded C++ VoD servers for target bitrates of 6–18 Mbps, with test configurations ranging from 1–8 disks. The Aspen and hand-coded versions consistently perform within 10% of each other. Aspen maintains a substantial advantage, however, by requiring far fewer lines of code and by easing maintenance.

6.3 Impact of Aspen-Specific Actions on Program Performance

To illustrate the impact on performance of Aspen's queuing and adaptive thread allocation decisions and flow-level concurrency support, Table 1 shows the five most expensive functions involved in those decisions, as reported by the `oprofile` Linux system profiler. The "Percentage Time Spent" column documents the percentage of the program execution time spent in the function "Function Name". As we can see, the net effect of all of Aspen's extensive runtime support for adaptive thread allocation and flow-level parallelism is less than 9%. Having said that, it is possible to lower this overhead still further. As an example, we are looking into ways of reducing the time spent in the `Queue::setAverageParseTime()`, which is what records the service time required by each module. This function is expensive for two reasons. First, it is called twice for every message dequeued. Second, it involves arithmetic operations and a possible iteration over a vector of C++ STL `bitset` instances. One possible optimization is to replace calls to this function with a simple bit-toggling operation for all modules that have been declared to be single threaded (via the use of the `force SingleThreaded` Aspen scheduling primitive).

6.4 Language Usability

Table 2 reports the number of lines of code required to implement the various high-performance web servers used in the benchmark tests. (Apache is not included, but its code length is far greater. Much of that code, however, relates to extensions not considered in this paper.) These numbers are as reported by the `sclc` source-code line counting tool and do not include whitespace or comment lines [4]. Where possible, we report the lines of code present under

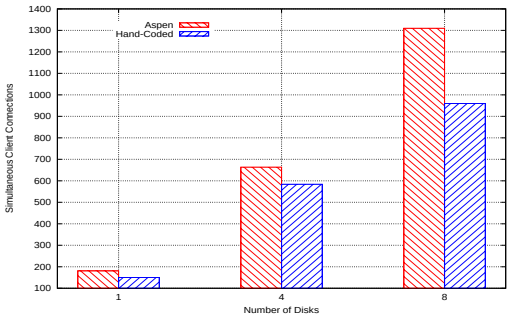

Figure 10. VoD server throughput with target bitrate of 1 Mbps.

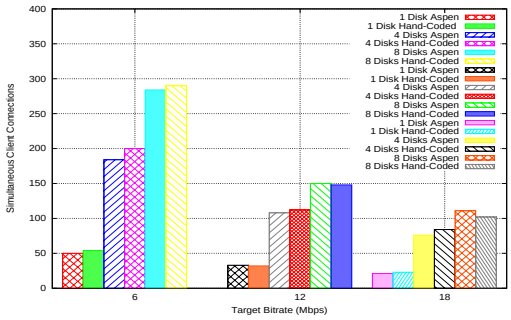

Figure 11. VoD server throughput with target bitrates from 6–18 Mbps.

Function Name	Function Description	Percentage Time Spent
Queue::setAverageParseTime	Updates values used by the adaptive thread allocator	2.7441
Queue::putCheck	Enqueue function	2.4875
Queue::waitingGet	Blocking dequeue	1.7408
LockedMap::put	Insertion into the hash map containing per-flow variables	0.7629
LockedMap::get	Getting per-flow variable information from hash map	0.3398

Table 1. Impact of Aspen flow-control and queuing decisions on performance.

Server Program	Language-Specific	User	System
Aspen (Web)	66	264	6014
Flash (Web)	NA	7226	NA
Flux (Web)	10	571	3297
Haboob (Web)	NA	2263	14100
Aspen (VoD)	88	339	5580
Hand (VoD)	NA	809	NA

Table 2. Lines of code used in implementing high-performance server applications described in Section 5, not counting white-space or comments.

three categories: "Language-Specific" lines of code are those that are part of the application but are unique to the implementation language, "User" refers to code that the user has written in order to implement the application (including language-specific portions), and "System" includes libraries and header files that are present with the application but are not part of a standard UNIX-like distribution. Of these categories, the "User" code is the most important since these are the actual parts that the application programmer must directly code. Note that Flash and Haboob do not have any lines of code under the "Language-Specific" category because they are written in C and Java, respectively. As seen, the web server written in Aspen requires 54–96% fewer lines of user-written code but still provides performance comparable to the others. Similarly, the video-on-demand server written in Aspen requires approximately 58% fewer lines than the hand-coded server to achieve the same functionality.

7. Related Work

As an infrastructure for efficient software engineering, Aspen relates to various domain-specific languages. The most closely related projects are Liberty, StreamIt, and Ptolemy [5, 32, 33], as all of these systems compose programs from concurrent modules that interact through explicit communication channels. StreamIt targets streaming media applications, Liberty targets architectural simulators, and Ptolemy targets hardware/software co-design of embedded communication systems. Modules in these systems react to events on their communication channels, as in Aspen. Aspen targets operating system-intensive codes and server applications running on general-purpose processors. Consequently, Aspen includes features not supported by those systems, such as dynamically-created copies of modules or the use of a hybrid event-driven/threaded execution model.

Burns et al. designed the Flux programming language, explicitly targeting server applications [6]. Flux expresses servers as graphs of action modules connected by well-defined interfaces, similar to Aspen's root modules. Flux also provides explicit support for atomicity and deadlock-avoidance in shared-address space programs; Flux consequently does not allow graphs with back edges. Aspen differs from Flux by providing language-level support for the action modules themselves. Further, since Aspen does not target programs with inter-module state-sharing, there is no support for atomicity nor any restriction on back edges in the graph.

Welsh et al. introduced the SEDA software architecture, which expresses a software system as a sequence of stages that communicate via queues [37]. SEDA also allows for the inclusion of resource controllers that allocate threads to the various stages. There are three major differences between the SEDA architecture and Aspen. First, SEDA is a methodology for program creation rather than a language; although SEDA can be applied to any language, none of those languages are naturally designed to provide support for concurrency and networking as Aspen is. Second, the resource controllers in SEDA are explicitly configured using programmer knowledge, whereas resource allocation in Aspen is automated using continuous inspection of load levels by the Aspen runtime. Finally, the individual stages in SEDA must be programmed in an event-driven fashion, whereas the modules in Aspen are written as straight-line serial code. Von Behren et al. observed that event-driven programming of SEDA stages was often difficult for their target applications [34]. In contrast, any event-driven interactions in Aspen are automatically managed by the Aspen runtime.

A variety of research has focused on the performance benefits of event-driven programming in network servers, with many works also integrating forms of multithreading [8, 25, 26]. However, the

models proposed by these works focus narrowly on operating system and library support for client-server applications, whereas Aspen integrates language support and allows more general communication mechanisms for applications based on message-passing or other communication schemes. The Capriccio project introduces the notion of logical threads, which expose a threaded interface to the programmer but internally utilize an event-driven scheduling loop created with the assistance of an intelligent compiler [35]. Unlike Aspen, though, that work does not provide language-level support for programmer productivity and only targets single CPU architectures.

Various domain-specific languages have targeted problems in the software engineering of networked services. Click [24] targets software routers, WebCaL [16] targets cache and communication policies for web caches, *nesC* [12] targets event-driven interactions in embedded sensor networks, and the Shangri-La and Nova projects target the efficient programming of the Intel IXP network processors [9, 13]. Wash/CGI provides a front-end to Haskell for safe programming of server-side web scripts [31]. These languages differ significantly in domain from Aspen and have correspondingly different structures, primitives, and runtime systems. For example, none of these languages deal with high-latency operating-system interactions or have any system for dynamic thread allocation.

Data and work-flow support at various levels has been proposed by others. Among the earliest of these are the Burroughs Work Flow Languages [7] and the IBM JCL [20]. These languages targeted a single family of proprietary systems, and only interactions of programs with external resources. Languages such as C&Co [1] operate on an explicitly transaction model and do not support Aspen's abstractions. Dataflow languages and languages for distributed computing target the expression of parallelism (e.g. [19, 17, 2, 18] or communication across processes (e.g. [19, 36, 27] but do not address the issues of changing communication primitives to accommodate and optimize for different execution environments, and of completely abstracting the communication structure from the logic of the program. Moreover, none of these languages have Aspen's support for storing independent flow state via associative memory, and none of these have Aspen's support for automatic thread allocation and destruction.

Libraries for coordination have been proposed, such as CORBA [3], MeldC [11] and Arjuna [28]. These, along with grid services provided by the Globus Grid Toolkit [15] and Globus Resource Specification Language RSL v1.0 [14] can be viewed as primitives targeted by the Aspen code generator.

8. Conclusions

Network services applications lend themselves well to task-level parallelism, but programming them can be very difficult because current programming languages are inherently sequential. This work introduces Aspen, a programming language that represents an application as a diagram of independent work-entities connected by explicit message queues, analogous to a flow chart. Aspen also includes primitives to support basic network services and includes a runtime that handles the intricacies of message communication between the modules. Aspen's runtime enables high performance by adaptively allocating *servlet* threads to module instances as required by varying load levels. Moreover, Aspen's unique features, such as per-flow variables, are shown to be useful and to incur almost no runtime overhead.

Aspen is tested using a web server and a Video-On-Demand (VoD) server application. These servers require far less user code than counterparts written using other programming methodologies or traditional languages, but nevertheless achieve performance comparable to or slightly better than those other servers. Aspen's adaptive thread allocation strategy allows superior performance in some cases because Aspen only allocates threads when they are needed, allowing servers written in Aspen to tolerate disk latencies without using an excessive number of threads.

References

[1] e. K. A. Forst and O. Bukhres. General purpose work flow languages. *Journal on Parallel and Distributed Databases*, (2):1–33, April 1995.

[2] K. Arvind and R. S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[3] R. Bastide, P. Palanque, O. Sy, and D. Navarre. Formal specification of corba services: experience and lessons learned. In *OOPSLA '00: Proceedings of the 15th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 105–117, New York, NY, USA, 2000. ACM Press.

[4] Brad Appleton. Sclc and Cdiff: Perl scripts for ClearCase. At `http://www.cmcrossroads.com/broadapp/clearperl/sclc-cdiff.html`.

[5] J. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, 4:155–182, April 1994.

[6] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of the USENIX 2006 Annual Technical Conference*, pages 129–142, June 2006.

[7] B6700 wfl primer, 1979. Available in the Gregory Publishing Company collection, Burroughs Manuals, 1976-1983, Charles Babbage Institute, University of Minnesota (Box 1, folder 3).

[8] A. Chanda. An efficient threading model to boost server performance. Master's thesis, Rice University, May 2003.

[9] 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 *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June 2005.

[10] D. Clitherow, S. Herzog, A. Salla, V. Sokal, and J. Trethewey. *OS/390 Workload Manager Implementation and Exploitation*. International Business Machines, May 1999.

[11] P. Dickman. Objects in large distributed applications (olda-ii). In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications (Addendum)*, pages 63–69, New York, NY, USA, 1992. ACM Press.

[12] 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 *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 1–11, June 2003.

[13] L. George and M. Blume. Taming the IXP Network Processor. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 26–37, June 2003.

[14] Globus resource specification language rsl v1.0, 2006. Last checked Jan. 5, 2006.

[15] Globus grid toolkit, 2006. Last checked Jan. 5, 2006.

[16] S. Gulwani, A. Tarachandani, D. Gupta, D. Sanghi, L. P. Barreto, G. Muller, and C. Consel. WebCaL: A Domain Specific Language for Web Caching. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, May 2000.

[17] J. Gurd and W. Bohm. Implicit parallel processing: SISAL on the Manchester dataflow computer. In *Proceedings of the IBM-Europe Institute on Parallel Programming*, Aug. 1987.

[18] N. Harvey and J. Morris. NL: A general purpose visual dataflow language. *Australian Computer Journal*, 12(1):2–12, 1996.

[19] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), Aug. 1978.

[20] Os/390 v2r10.0 mvs jcl reference, 2000. Document number GC28-1757-09.

[21] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable Stream Processors. *IEEE Computer*, pages 54–62, August 2003.

[22] H. Kim, S. Rixner, and V. S. Pai. Network Interface Data Caching. *IEEE Transactions on Computers*, 54(11), November 2005.

[23] J. R. Larus and M. Parkes. Using Cohort Scheduling to Enhance Server Performance. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 103–114, June 2002.

[24] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek. The Click modular router. *TOCS*, 18(3):263–297, August 2000.

[25] E. Nahum, T. Barzilai, and D. Kandlur. Performance Issues in WWW Servers. In *Proceedings of the SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 216–217, May 1999.

[26] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An Efficient and Portable Web Server. In *Proceedings of the USENIX 1999 Annual Technical Conference*, pages 199–212, June 1999.

[27] R. Salama, W. Liu, and R. S. Gyurcsik. Software experience with concurrent c and lisp in a distributed system. In *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 329–334, New York, NY, USA, 1988. ACM Press.

[28] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the arjuna distributed programming system. *IEEE Softw.*, 8(1):66–73, 1991.

[29] The Standard Performance Evaluation Corporation. SPECWeb99 Benchmarks. At `http://www.spec.org/osg/web99/`, 1999.

[30] The Standard Performance Evaluation Corporation. SPECWeb2005 Benchmarks. At `http://www.spec.org/osg/web2005/`, 2005.

[31] P. Thiemann. An Embedded Domain-Specific Language for Type-Safe Server-Side Web-Scripting. *ACM Transactions on Internet Technology*, 5(1):1–46, February 2005.

[32] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[33] M. Vachharajani, N. Vachharajani, D. A. Penry, J. A. Blome, and D. I. August. Microarchitectural Exploration with Liberty. In *Proceedings of the 35th Annual International Symposium on Microarchitecture*, pages 271–282, November 2002.

[34] J. R. von Behren, E. A. Brewer, N. Borisov, M. Chen, M. Welsh, J. MacDonald, J. Lau, S. Gribble, and D. Culler. Ninja: A Framework for Network Services. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 87–102, June 2002.

[35] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable threads for internet services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, October 2003.

[36] P. H. Welch. An occam approach to transputer engineering. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 138–147, New York, NY, USA, 1988. ACM Press.

[37] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.