AESOP: Expressing Concurrency in High-Performance System Software

Dries Kimpe, Philip Carns, Kevin Harms, Justin M. Wozniak, Samuel Lang, Robert Ross
Mathematics and Computer Science Division
Argonne National Laboratory, Argonne, IL 60439
{*dkimpe,carns,harms,wozniak,lang,rross*}*@mcs.anl.gov*

***Abstract*—High-performance computing (HPC) and distributed systems rely on a diverse collection of system software to provide application services, including file systems, schedulers, and web services. Such system software services must manage highly concurrent requests, interact with a wide range of resources, and scale well in order to be successful. Unfortunately, no single programming model for distributed system software currently offers optimal performance and productivity for all these tasks. While numerous libraries, languages, and language extensions have been developed in recent years to simplify parallel computation, they do not address the challenges of distributed system software in which concurrency control involves a variety of hardware and network devices, not just computational resources.**

In this work we present AESOP, a new programming language and programming model designed to implement distributed system software with high development productivity and run-time efficiency. AESOP is a superset of the C language that describes blocks of code to be executed concurrently without dictating whether that concurrency will be provided by a threading, event, or other model. This decoupling enables system software to adjust to different architectures, device APIs, and workloads without any change to the core algorithm implementation. AESOP also provides additional language constructs to simplify common system software development tasks.

We evaluate AESOP by implementing a basic file server and comparing its performance, memory efficiency, and developer productivity with several thread-based and event-based implementations. AESOP is shown to provide competitive performance to traditional distributed system software development models while at the same time reducing code complexity and enhancing developer productivity.

I. INTRODUCTION

High-performance computing and distributed systems rely on a diverse collection of system software to provide shared services for applications. Examples include file storage, scheduling, security, checkpointing, web services, membership management, and fault detection. Today's systems demand that these services scale to hundreds of thousands of concurrent clients, with millions of clients expected in the near future. Developing system software for this environment is a complex task because of a number of competing software engineering requirements:

- Scalable orchestration of concurrent client and device activity
- Maintainable code for complex distributed algorithms
- Portability and performance across diverse architectures

Further complicating matters is the fact that system software services not only must interact with multiple hardware devices concurrently but must do so using a wide assortment of interfaces and progress models. Popular network transports, disks, and local databases may present a simple set of blocking functions, fully asynchronous callbacks, or any one of a number of API models in between. In some cases, the most efficient API is platform specific. This diversity makes it difficult to build algorithms that are portable and yet capable of leveraging each component in an optimal manner.

The two most common models for managing asynchronous activity in system software are multithreading and event-driven architectures. Examples of both can be found in various large-scale production services. The Apache web server advocates a multithreaded mode of operation for high-performance deployments [1]. In contrast, the memcached object caching service uses an event-driven model with threads used to drive concurrent event loops [2].

The multithreading model is widely used and well understood in the development community. It produces a natural control flow in algorithm implementations because each thread executes sequentially and issues simple blocking device operations. Thread scheduling and stack management are handled by the operating system. Building a multithreaded service raises a number of challenges, however. The first is determining how to provision thread resources. Threads can be assigned per client, per request, or per underlying device operation; they may be created on demand or allocated from an existing pool. Each technique or combination of techniques presents tradeoffs in scalability, performance, and readability. Yet it is difficult to transition between techniques during the development process. The second challenge is the disconnect between the number of threads needed to express logical concurrency and the number of threads needed to make optimal use of local resources. For example, the optimal number of concurrent operations for a storage device will be distinct from the optimal number of threads per processor core. This disconnect can cause a seemingly simple thread model to evolve into a more complex framework when taking local resource-scheduling

constraints into account.

Event-driven models can be used to address some of the common challenges in thread efficiency. Such models strive to unify the management of asynchronous activities, so that the completion of any storage or network event triggers execution of the next servicing step through a centralized event handler. This architecture provides greater control over asynchronous activity and avoids the use of operating system threads to express logical concurrency. Any number of client requests can be tracked and serviced without explicitly allocating a thread to each request. This model has drawbacks as well, however. Event-driven architectures break the logical control flow of software algorithms into disjoint segments at each point where the algorithm interacts with an asynchronous device or operating system service. In effect, it turns the single linear control flow for an algorithm into a state machine. Because these states are invoked from a centralized event handler, their interfaces must be consistent and generalized, making it difficult to verify design constraints. State transfer across event handlers has to be managed manually by the programmer, a process often referred to as *stack ripping* [3]. Pure event-driven models also do not directly address the requirement of multicore architectures or high-performance peripherals that require multiple threads to maximize throughput. Event-driven servers therefore tend to evolve during the development process to take on characteristics of both event-driven and multithreaded services in order to compensate for these issues.

In general, no clear-cut metric can be used to identify the best concurrent programming model for a given system software project, yet this decision affects every aspect of the design, from algorithm implementation to device management. To help address this problem, we propose a new programming language and programming model, AESOP, that targets the software engineering challenges associated with developing distributed system software. Its goal is to improve programmer productivity by presenting a convenient model for developing distributed system software. The model isolates the developer from how concurrency is managed, but maintains execution efficiency through its ability to map AESOP code to the concurrency model preferred by the system or device.

AESOP is a superset of the C language. It describes blocks of code that can be executed asynchronously or concurrently, without dictating whether those blocks must be implemented with threads, and without dictating the asynchronous progress model to be used by any underlying devices. Because the core algorithm description is decoupled from these architecture-specific details, the run-time system can be tuned to match different system architectures, different device APIs, and different workloads without any change to the core system software algorithms. Unlike event-driven architectures, AESOP preserves readable control flow

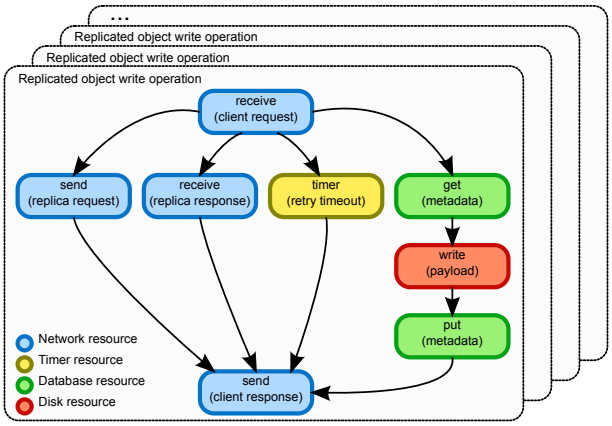


Figure 1: Concurrency in file system server.

regardless of how many steps or concurrent devices are utilized in a given algorithm. AESOP also offers functionality that is not available in a traditional multithreaded or event-driven framework – for example the ability to cleanly cancel outstanding operations.

The contributions of this paper can be summarized as follows:

- Description of a new programming model and language, AESOP, for use in the development of distributed system software
- Definition of new language constructs to help manage highly concurrent execution
- Case study comparing AESOP to five popular multithreaded and event-driven architectures, using a simple distributed service as an example
- Quantitative analysis of the code complexity of the architectures used in the case study

The remainder of this paper is organized as follows. Sections II and III describe the challenges of distributed system software development and related work in addressing those challenges. Sections IV and V present the AESOP programming model and our implementation. Section VI describes a simple distributed service and uses it as a case study to compare AESOP with other models in terms of performance, memory efficiency, and productivity. Section VII summarizes our findings and proposes avenues of future work.

II. CONTEXT

Large-scale distributed and parallel systems rely on distributed system software to provide scalable access to persistent services. For example, a parallel file system aggregates distributed storage resources provided by multiple servers into a unified name space. Each individual service program expects many concurrent accesses from multiple clients, and, in turn, launches multiple overlapping operations such as network transmissions or disk drive operations.

A. Motivation: Concurrency in File System Services

Consider the file server depicted in Figure 1. An operation starts when the client issues a request to the server, triggering the top receive. The server then simultaneously issues four concurrent operations: a send to a peer server to forward data to a replica, a receive to accept the response from the replica server, a timer call to ensure the request completes within a reasonable time, and a sequence of data and metadata operations to perform the local write operation. If all operations complete before the timer expires, the timer is cancelled, and the result is posted to the client. However, if the timeout is reached before the other tasks complete, all unfinished operations are cancelled and an error is returned to the client.

This example illustrates that concurrency arises not only from simultaneous client access but also from the steps required to service each request. Ideally, programming models should help manage this concurrency, enhancing developer productivity. Note that this is a simplified example. A production-quality file system server additionally requires pipelining, failure handling, and retry functionality, all of which further complicate its implementation.

B. Device Interfaces

A programming model designed to ease managing asynchronous device operations must take into account the broad array of relevant underlying devices and their interface models. For a device exporting a blocking interface, function calls do not return until the request described by the call is completed. Concurrency is typically achieved by calling multiple functions simultaneously from multiple threads. However, for scheduling and efficiency reasons (described in Section I), high-performance storage and networking APIs typically provide nonblocking interfaces to achieve I/O concurrency.

Nonblocking interfaces follow an *event-driven model*, where a request for an I/O operation is made to the device through a function call, which returns without waiting for completion (the function does not block). The program receives completion of the I/O operation (the event) through other notification paths, such as through a polling function or asynchronous function callbacks. Nonblocking interfaces achieve concurrency by enabling the posting of multiple I/O operations without waiting for their completion.

Network sockets offer multiple potential asynchronous APIs that follow the select/poll paradigm or an operating system specific variant. InfiniBand offers a post/poll interface to send/receive queues through the `libibverbs` API [4]. Myrinet provides a rich API through the MX interface [5] that offers asynchronous send/receive operations with test, probe, and callback registration. The IBM Deep Computing Messaging Framework (DCMF) [6] primarily offers nonblocking calls for send/receive and one-sided put/get communication. MPI-2 [7] offers nonblocking messaging calls for point-to-point communication.

Disk operations also have a range of nonblocking access methods. POSIX AIO [8] offers multiple models for controlling asynchronous I/O, including the ability to supply a signal or callback function for completion notification.

Databases are commonly used for file system metadata, where efficiency is critical. Berkeley DB [9] does not provide a nonblocking API, but wrapper libraries have been used to achieve this. Postgres offers an asynchronous interface [10] in its C API.

III. Related Work

A number of programming languages and programming models have been developed to express application-level concurrency and efficiently leverage multicore architectures. Examples include OpenMP [11], X10 [12], Unified Parallel C [13], Chapel [14], Fortress [15], CUDA [16], and OpenCL [17]. Such languages allow developers to annotate code blocks that should execute in parallel, including constructs for synchronization and sharing variables. While many of these constructs share principles with AESOP, they were developed with the goal of coordinating computation on multiple cores at the application level, rather than coordinating asynchronous activity on multiple I/O devices at the system software level. As a result, these languages optimize for maximum use of available CPU resources, whereas AESOP optimizes for maximum use of underlying devices that may or may not require significant CPU activity. AESOP in general has the closest similarity to OpenMP but differs in that AESOP does not create or require the use of threads. The level of AESOP concurrency is not correlated with the number of CPU cores available, though AESOP can utilize multiple cores (if available) to improve latency and concurrency. The other main differentiation is that AESOP also offers language support for cleanly cancelling outstanding concurrent executions paths.

Grand Central Dispatch (GCD) [18] is a popular programming model introduced by Apple Inc. that allows developers to describe tasks to be executed concurrently. GCD manages an implicit thread pool to schedule and execute the tasks. If the tasks are CPU bound, then GCD may match the number of threads to the number of CPU cores. If the tasks block on device activity, then it may instantiate a larger number of threads. The primary advantages of GCD are that it simplifies multithreaded programming and automatically adjusts concurrency according to the workload and the system architecture. Unlike AESOP, however, GCD always uses threads to achieve concurrency and provides no general framework for integration with device APIs that provide more scalable asynchronous interfaces. Moreover, GCD does not support cancellation of tasks.

In addition to programming models and languages, several support libraries have been developed to aid in portably

managing a large number of asynchronous events. Examples include libevent [19] and libev [20]. Such libraries provide a unified API for managing multiple kinds of events, particularly on sockets or file descriptors. While such libraries solve an important technical problem related to service efficiency, they do not address the programmability aspect of event-driven concurrent programming. The developer is left the task of fitting their algorithm logic to the event-driven model.

The AESOP language described in this work builds on lessons learned from the event-driven state machine model used in the PVFS file system [21]. In PVFS state machines (SMs), every client and server operation is expressed as a state machine using a language that is based on C with extensions to describe service states and the transitions that link them. A state is simply a C function. Each state function can optionally end by submitting an asynchronous network, disk, or timer operation through an API that provides unified notification of completion events for each device. When one of these operations completes, a state machine engine maps the operation back to the appropriate state machine and executes the next state function, depending on the outcome of the asynchronous operation. Multiple state machine instances can be active simultaneously, allowing PVFS to track the state of an arbitrary number of concurrent requests and make progress on them without the overhead of explicit threading.

The PVFS state machine model is essentially a formal framework for event-driven programming. This approach has proved successful in achieving high performance in large-scale production HPC environments [22], but it shares many of the same developer productivity challenges found in other event-driven models; it disrupts control flow, requires programmers to perform stack ripping, and utilizes only a single compute core to execute event handlers. Moreover, PVFS SM code does not resemble traditional C code, making it difficult to reason about many development challenges. These drawbacks introduce a learning curve for researchers and an additional maintenance workload for maintainers. In this work, we seek to learn from the lessons of PVFS SM approach and present a programming model that provides at least as much functionality but also allows file system developers to write algorithms using a more traditional code organization.

IV. PROGRAMMING MODEL

This section describes the AESOP progamming language and its associated program model.

A. Parallel Branches

Concurrency is expressed in AESOP through the use of *parallel branches*, or *pbranches*. A parallel branch groups a list of statements and supports branch-scoped variables, much like a regular function. Within a pbranch, statements are executed sequentially, as in a normal C program. When multiple pbranches are active, however, the AESOP language enforces sequential execution only within the scope of a single pbranch. Statements from other pbranches might interleave execution or might execute concurrently (using a thread for example).

AESOP can synchronize pbranches by using the `pwait` construct. A pwait is similar to the barrier implicit in many OpenMP directives (such as parallel for), in that no statements following the pwait will execute unless all the pbranches it encloses have completed. AESOP also supports *lonely pbranches*, that is, pbranches created outside a pwait. A lonely pbranch resembles a POSIX detached thread in that lonely pbranches cannot be synchronized with other user code.

By default, variables in AESOP are shared between pbranches. This configuration is similar to OpenMP, where variables are shared between multiple threads unless indicated otherwise. Marking a variable as `pprivate` gives each pbranch a private shadow copy, initialized by using the shared instance when the branch is created. Variables declared within a pbranch are, adhering to the C scoping rules, necessarily private.

A pbranch can request the termination (or cancellation) of all other pbranches within the same pwait by calling the `aesop_cancel_branches` function. This function returns as soon as the other pbranches have been marked for cancellation; it is asynchronous in that it does not wait for the other pbranches to exit. If a pbranch is in a function call when the cancellation signal arrives, that function will be notified of the cancellation, enabling the function to take whatever steps are necessary to cancel the operation and return a suitable return code. Further function calls within the cancelled pbranch will detect the cancellation state upon entering the function and react accordingly. By design, it is not possible for a pbranch to clear its cancellation flag.

AESOP's cancellation model differs from others in that the language itself takes care of cleanly cancelling pending operations, transparently emitting the needed code to do so. For example, if a device exposes an asynchronous API, AESOP will automatically call the corresponding `cancel` function when a pbranch waiting for the completion of an operation is cancelled. More details about AESOP's cancellation model are presented in Section V. Many other libraries aimed at concurrent or asynchronous execution do not support cancellation or offer only limited support. In Grand Central Dispatch, once a code block is dispatched to a queue, it can no longer be removed or cancelled. Likewise, OpenMP does not support cancelling helper threads in parallel for and other concurrency constructs. In both cases, the recommended practice is for the programmer to repeatedly check a cancellation flag. This becomes especially cumbersome if other functions are called from within the concurrent code, since now those functions must have access to the cancellation variable as well.

While the POSIX thread API supports a number of different cancellation modes, using them in production software is not trivial. In the `asynchronous` cancellation mode, depending on the system, thread execution might be halted at any point, making it hard for the programmer to properly track cleanup state. In the default `deferred` mode, cancellation is ignored by all but a small list of special functions (termed `cancellation points`). Many common C functions, such as `fread`, may or may not, depending on the system, support cancellation. Most do not. Thus, a portable program cannot call any of these functions if timely cancellation is required.

B. `aesop` Functions

AESOP extends the C programming language with a new function type: *aesop* functions. The `aesop` qualifier is part of the type. A pointer to a regular C function is not compatible with a function pointer to an `aesop` function with the same function arguments. `aesop` functions differ from regular C functions in that only `aesop` functions can contain pbranches. In addition, `aesop` functions support cancellation.

Functions can be brought into the AESOP model by adding the `aesop` qualifier to the function declaration. However, `aesop` functions can be called only by other `aesop` functions. Thence, the starting function of an AESOP program must be an `aesop` function. AESOP libraries form an exception to this rule. To simplify calling AESOP libraries from C and other languages, one can generate C function stubs for an `aesop` function. These stubs provide either synchronous or asynchronous bindings to AESOP libraries.

An active `aesop` function in a pbranch receiving a cancellation request will be notified of the cancellation. The effect of this notification and the way the function reacts to this request depend on the implementation of that particular function. Likewise, any subsequent `aesop` call initiated in that pbranch will learn of the active cancellation request. While this is not enforced by the language, a proper `aesop` function should return as soon as possible when a branch is in a cancelled state.

The intention is that functions that are not completely CPU-bound (i.e., cannot efficiently fully consume a thread) are to be `aesop` functions. AESOP detects when a `aesop` function can no longer make progress (for example, because the function is waiting on an external event or device). When this happens, execution will continue in another branch.

Because of the pbranch construct, the AESOP language does not describe or require a thread concept in order to expose concurrency. In fact, instead of promising explicit concurrent execution for pbranches, the language guarantees that, when one or more pbranches within the program are ready to execute, at least one of them will make progress. For example, in a pwait containing two branches, if one branch should stall on a `aesop` function (for example waiting for I/O), execution will proceed in the second pbranch provided that pbranch is ready.

```
1  aesop void doReplicaWrite (...) {
2    pwait {
3      pbranch { concurrent_step_1 (); }
4      pbranch { concurrent_step_2 (); }
5      pbranch { concurrent_step_3 (); }
6    }
7  }
8  aesop void replicaWrite (...) {
9    pwait {
10     pbranch {
11       aesop_timer (TIMEOUT);
12       ae_cancel_branches ();
13     }
14     pbranch {
15       doReplicateWrite (...);
16       ae_cancel_branches ();
17     }
18   }
19 }
20 aesop void write (...) {
21  pwait {
22    pprivate int i;
23    for (i=0; i<REPLICAS; ++i)
24     pbranch {
25       replicaWrite (i, ...);
26     }
27    }
28 }
```

Listing 1: Aesop code example.

In effect, the AESOP programming model provides for multiplexing multiple pbranches onto a single thread or core, similar to application-implemented user space threading with voluntary yielding. This does not preclude true multithreading; AESOP is fully thread-safe and multiple OS threads can execute within a single AESOP program or the AESOP runtime library. In that case, if more than one pbranch becomes ready for execution, multiple threads will be used to execute those branches concurrently.

C. Example Aesop Code

Listing 1 shows a common AESOP code pattern. Network servers, having a finite set of resources, must protect against unbounded resource consumption by misbehaving or failing peers. Typically, an upper limit is placed on the time resources are dedicated to a request, cancelling the request when the allocated time is exceeded.

In the code example, `write` concurrently forwards the write operation to `REPLICAS` replicas. This is implemented using a `for` loop containing a pbranch. Since the pbranch statement does not wait for the completion of the branch, the next iteration of the `for` loop can start before the previous one completed. In `replicateWrite`, a common pattern for easily handling time-outs can be seen. Two pbranches are created, one to execute the work and the other to bound the maximum time the other branch is allowed to execute. When one of the branches completes, as a last task it cancels the other branch. Thus, if `doReplicateWrite` (line 15) completes first, the operation completed within the allocated time frame, and the timer (line 4) will be cancelled. However, if line 15 takes longer than `TIMEOUT`

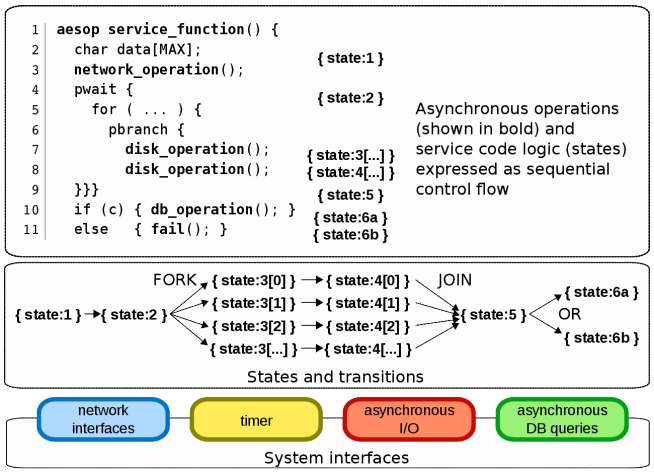


Figure 2: Model of Aesop usage.

to complete, the timer (line 11) will complete first, causing the ongoing `doReplicaWrite` operation to be cancelled. The `ae_cancel_branches` call internally synchronizes, preventing both branches from cancelling each other. Note how `replicaWrite`, in a straightforward and reusable manner, adds a time-out to the existing `doReplicaWrite` call.

The short code fragment from Listing 1 implements the pattern shown in Figure 1, in which a number of concurrent write operations are started; each consists of three tasks that can be executed concurrently, with each write operation protected by a timeout.

V. Implementation

We implemented the AESOP language using a source-to-source translator. It processes AESOP code and rewrites AESOP-specific functionality using standard C constructs, emitting fully compliant C code. AESOP code can therefore be deployed on any system that provides a suitable C compiler. Since AESOP implements an extension to the C language, all valid C programs are also valid AESOP programs, and all C features can be used within an AESOP program. Currently, the translator needs to rewrite only `aesop` functions; all other code is passed through unmodified. Figure 2 shows how an `aesop` function, `service_function`, is translated by the current version of the AESOP translator. As a first step, all `aesop` function calls within the function body are located. Based on the location of the `aesop` calls, the function is split into a set of code blocks (states), where each code block contains only nonblocking (traditional C) function calls. The calls themselves are rewritten in an asynchronous form. For example, `network_operation` becomes `post_network_operation`. This post function becomes the last statement of the preceding code block, where the first statement following the `network_operation` call becomes the first statement of the next block. As part of the translated `post_network_operation` call, AESOP injects a callback argument that indicates where execution should continue once the operation completes. As soon as the operation is posted, `service_function` returns, unwinding the stack. Depending on the state and structure of the program, the stack might unwind all the way back to the starting point of an AESOP program. When this situation happens, AESOP will try to make progress on any pending operation, until one or more operations complete. Once an operation completes, AESOP invokes its callback function to resume execution of the next code block.

As an optimization, AESOP recognizes when an operation completes before the post function returns. When this happens, no stack unwinding occurs, and execution immediately continues with the next state in the function.

Because variables local to the function scope are stored on the stack and are destroyed when returning from the function, AESOP, as part of the rewriting process, moves all function scope variables (and function arguments) into a context structure on the heap, transparently rewriting variable references to account for this change. A pointer to this context is passed to the callback when the execution of the function resumes.

AESOP keeps track of pending operations within a given context. When a request is made to cancel concurrent branches, AESOP will call the correct cancellation function for each pending operation in the current context. To do so, AESOP needs to know, for each `aesop` call, how to cancel the operation. For functions translated by AESOP (those marked by `aesop`), AESOP internally provides the cancel methods.

However, for operations managed by an existing, external library (for example, AIO), this information needs to be provided to AESOP. This is done by creating a *resource*. Resources are typically used to provide an AESOP binding on top of an existing C library. A resource exports an AESOP API consisting of regular and `aesop` functions. It internally registers a set of hooks that can be called by AESOP to make progress, test, or cancel a `aesop` function exported by the resource API. Progress can be made by providing a polling function, which will be called by AESOP when no other work is available. However, AESOP does not enforce a progress model on the resource. Some resources do not define a polling function, typically because the resource or the underlying C library can make progress independently.

VI. Evaluation

To evaluate AESOP, we implemented a simple TCP network server using traditional techniques and libraries and compared the resulting implementations with an AESOP implementation of the server. We evaluated each server implementation using three criteria: run-time efficiency, memory efficiency, and programmer productivity.

Each of the servers implements the same request protocol and recognizes four request types: write, read, write-null, and read-null. In the write case, the server receives a request containing the file name, access size, and a data payload. It creates the specified file, writes the data (using `O_DIRECT`), and responds to the client. The read case is similar to the write case, except that data is read from the specified file and sent to the client. Write-null and read-null are variants of write and read that perform no file access. The write-null payload is simply discarded by the server, while the read-null payload transmitted to the client consists of uninitialized memory.

Each server uses the same fundamental coding style to the degree possible. One server is implemented in the AESOP language, while all other servers are implemented in C. The pthread library was used where thread support was needed; the libev [20] library was used where event handling was needed. A short summary of each server implementation follows.

AESOP: The AESOP server is implemented by using the AESOP programming language. A *lonely pbranch* is used to service each client. All operations for a client are handled within a single pbranch. The socket and file operations are performed with `aesop` functions provided by the AESOP standard library. On this system, the underlying socket resource uses nonblocking sockets in an event driven model, with up to 12 threads driving the event loops. The file resource uses synchronous I/O calls and a thread pool with 4 threads.

Thread-per-client: The thread-per-client server spawns a dedicated thread for each client connection. All requests for a given client are handled within the same thread. This model uses synchronous socket and file I/O functions.

Thread-per-client-nb: The thread-per-client-nb server is identical to the thread-per-client server except that it uses asynchronous socket calls in place of synchronous socket calls. We implemented this version to isolate the performance impact of asynchronous socket access when all other factors remain equal.

Thread-per-operation: The thread-per-operation server uses an event loop to watch all client connections for activity. When a new request is available, a thread is spawned, and the request is handled completely from within that thread.

Thread-pool: The thread-pool server is similar to the thread-per-operation server in that it uses an event loop to watch client connections for activity. However, instead of creating a new thread to service each request, the requests are serviced by a fixed-size thread pool.

Event: The event server uses an event loop not only to detect active connections but also to service them. Each request-processing step is executed from an event loop callback function. The event server uses asynchronous sockets and asynchronous file I/O. Note that although this implementation does not explicitly create or use threads, the operating system can internally still use multiple threads to drive both the network and disk.

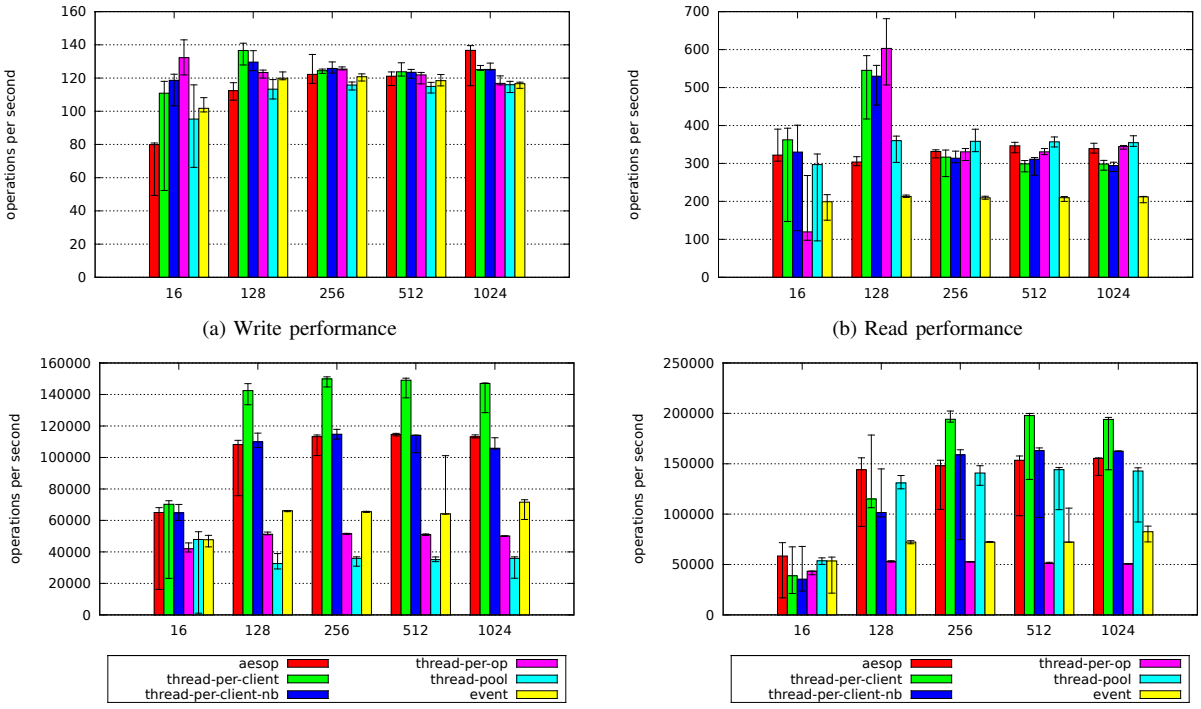
The servers were all evaluated using the same client test harness. The client test harness is a C program that uses TCP sockets to send messages to the server. It uses MPI to coordinate processes and generate a concurrent workload. Each client process records the total amount of time taken to execute its workload, beginning before the initial connection and ending after receipt of the last acknowledgment. The time taken by the slowest client is considered to be the aggregate run time. Each client also tracks the time needed to service each individual request in order to analyze individual request latency. The client test harness was configured to generate requests of size 4 KiB in all cases. Each client process performed 16 requests in the write and read tests and 4,096 requests in the write-null and read-null tests.

All experiments were executed on the Fusion cluster in the Argonne Laboratory Computing Resource Center. Fusion is an IBM iDataPlex dx360 M2 system with a QDR Infiniband interconnect featuring 320 compute nodes, consisting of two Intel Nehalem 2.6 GHz Xeon processors and 36 GB of RAM. Each compute node also has a single SATA 7200 RPM hard disk for local scratch storage, which was used for all disk I/O in the experiment. Client/server communication was done over the IB network using IPoIB. We instantiated 16 clients processes per phyiscal node. The tests were executed on 65 nodes; one server node and 1-64 client nodes.

A. Run-time Efficiency

Figure 3 shows, for each of the four test types, the median, minimum, and maximum number of requests completed per second. Each test type was executed 5 times, with the number of clients ranging from 16 to 1,024. In Figure 3a we see that AESOP does not perform as well as the other servers for a small number of clients (delivering 79 operations per second at the smallest scale, versus 132 ops/s for the thread-per-op server). However, AESOP is the fastest server at the largest scale (136 ops/s versus 125 ops/s for the nearest competitors in thread-per-client and thread-per-client-nb). When reading (Figure 3b), AESOP performs more favorably at small scale. At the largest scale, AESOP completes the test with 339 ops/s versus 354 ops/s for the fastest server (thread-pool), a 4.5% difference. The event server performs particularly poorly in all cases, ultimately running the largest-scale test with only 212 ops/s. The small-scale results for AESOP may indicate that additional tuning is needed to improve latency for small test runs. The issue is likely isolated to the write path of the file I/O resource in the AESOP standard library, as we see asymmetric results in the read and write tests for AESOP.

AESOP is competitive with the other implementations except for the thread-per-client server in both the write-null (Figure 3c) and read-null (Figure 3d) evaluation. One notable difference in the two implementations, however, is



(a) Write performance

(b) Read performance

(c) Write-null performance

(d) Read-null performance

Figure 3: Run-time performance for each test case.
(X-axis is the number of client instances, each graph has a distinct Y-axis.)

that the thread-per-client server uses blocking socket operations, whereas the AESOP socket resource uses nonblocking operations. Based on this observation, we implemented the thread-per-client-nb server to isolate the impact of nonblocking socket operations on performance. The thread-per-client-nb implementation is identical to the thread-per-client implementation except that each socket uses nonblocking operations and polling to transmit and receive data. As seen in these tests, the use of nonblocking operations slows the thread-per-client server to the point that it is practically equivalent to the AESOP server at scale, suggesting that asynchronous socket operations simply do not perform as well as synchronous socket operations on this system.

Another notable observation in these graphs is that the AESOP server is competitive at small scale and, in fact, is the fastest implementation in the 16-client-process read-null test and nearly the fastest in the 16-client-process write-null test. These results support the observation that poor AESOP performance at small scale is likely a tuning flaw in the AESOP implementation of I/O functionality, rather than a fundamental programming language problem.

The test client also measures the latency of each individual request and then computes the minimum and maximum latency, the first quartile latency, and the third quartile latency. Figure 4 shows the request latency for 1,024 clients for the write test. Each box represents the first and third quartiles, and the whiskers are the minimum and maximum

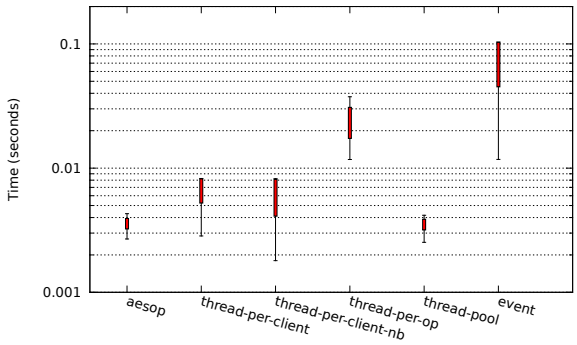


Figure 4: Write latency (1,024 clients).

values. From the figure, one can see that the AESOP server implementation exhibits similar latency results when compared to the other implementations. The event server variant shows significantly higher latency, however. This is likely an artifact of the Linux asynchronous I/O implementation. As the latency numbers for the other request types show similar results, only the write test results are shown.

B. *Memory Efficiency*

Another aspect of the overall performance is the memory efficiency of each server implementation. In this section we compare the memory usage of the AESOP server with that of the other server implementations. The memory utilization of each server was captured during the run-time performance

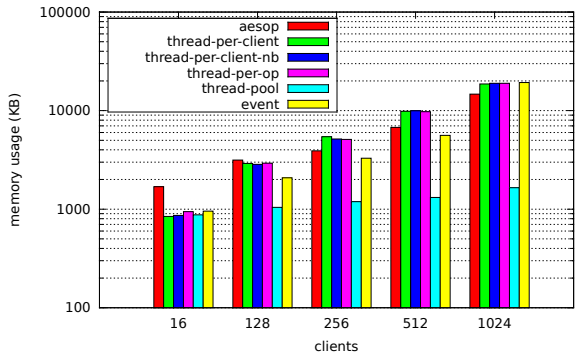


Figure 5: Write memory usage.

experiments. We recorded the VmHWM stat from the server when the client test was completed. The VmHWM stat is a Linux-specific metric that represents the peak resident set size (RSS) of an executable, where RSS corresponds to the amount of paged-in memory used by the executable.

In Figure 5 we see that the thread-pool server manages to limit the memory usage even though the client work load increases. This result can be explained by the design of the thread-pool; by limiting the number of active threads used to service client connections, it also limits the number of concurrent requests. For the other server implementations, memory usage increases accordingly as the number of concurrent requests increases. Although the AESOP server cannot match the thread-pool server in terms of memory usage, it does compare favorably with the thread-per-client and thread-per-op servers. Note that the thread-per-client and thread-per-op models consume virtual memory at a much larger rate because of the amount of virtual memory allocated to each thread stack. We chose not to evaluate this metric, however, because the resident memory seems to be a more relevant metric in practice. Since the memory usage for remaining tests (read, read-null, and write-null) was similar, only the write test results are shown.

C. Productivity

We infer programmer productivity by measuring code complexity. Table I compares the code complexity of each server implemenation using McCabe Cyclomatic Complexity (CC) [23], Modified McCabe Cyclomatic Complexity (mod. CC) using *pmccabe* v2.6 by Paul Bame, and Source Lines of Code (SLOC) using *sloccount* v2.26 by David A. Wheeler.

Table I: Code complexity analysis

	CC	mod. CC	SLOC
AESOP	16	11	179
thread-per-client	17	12	182
thread-per-client-nb	17	12	184
thread-per-op	22	17	249
thread-pool	32	26	313
event	28	23	341

In order to simplify the comparison, all four servers had no error handling except for assertions on expected return codes. The protocol definition as well as helper functions for sending and receiving were not counted in any of the implementations, since these were similar in all four.

The AESOP and thread implementation are similar in terms of complexity, with the slight increase in the thread model due to function calls needed to create and join threads.

The thread pool and event model implementation are both much more complex than the thread or AESOP model. An additional complexity of the event model which is not captured by these metrics is that control flow is not preserved across the processing of a given request. For example, servicing a write operation requires five disconnected event handlers.

Another productivity aspect that is not well captured by the complexity metrics is how easily our example server can be retargeted to a new architecture or system. In our initial testing, we wanted to tune the AESOP server implementation to perform well on a few test systems. During this testing we tried different strategies (event based, threads, and hybrid models) for sending and receiving network data and experimented with multiple threads driving AESOP. These changes were done within the AESOP run-time library and never required changing the AESOP server test code. This type of flexibility allows experimentation to determine the best tuning strategy for any given system without having to redesign the core software algorithms.

The example server used for evaluation is simple and differs from real-world code by the absence of error handling and time-out handling. As described in Section IV-C, AESOP provides powerful primitives to simplify these activities. The same cannot be said of the event or threaded models, since neither model offers any help in cancelling an outstanding operation.

VII. CONCLUSIONS

In this work we have introduced a new programming model and programming language, known as AESOP. AESOP was designed with the explicit goal of supporting distributed system software development. Its primary advantage is that it allows system software developers to express concurrency without dictating the mechanism used to provide that concurrency. It also provides a number of language constructs that simplify common system software development patterns. Numerous languages and language extensions have been developed in recent years to simplify parallel computation, but AESOP is unique in that it addresses the challenges of distributed system software in which concurrency control goes beyond computational resources to include a various hardware and network devices.

Using a simple network server for evaluation, we demonstrate that AESOP offers competitive, and in some cases superior, run-time performance to the most common system

software concurrency models in use today. We expect AESOP to continue to perform well at even larger scales than those evaluated in this study because it does not rely on mapping concurrent execution paths to dedicated operating system threads. When considering programmer productivity, AESOP performs equally well, mainly because it has been specifically designed to implement highly scalable distributed services. It reduces code complexity while also offering advanced primitives such as cancellation of execution paths that are not available in any other system software development environment.

We will continue to evaluate and evolve the AESOP language as we use it to implement future production HPC storage services. As part of this work we plan to extend the language to include additional features. One example, which exists in prototype form already, is a remote procedure call framework that uses AESOP to manage concurrent communication, timeout, and retry functionality in a transparent manner. We also plan to provide enhanced functionality for debugging AESOP functions. AESOP, and the code used to evaluate its performance, is available online at http://trac.mcs.anl.gov/projects/aesop.

ACKNOWLEDGMENTS

This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357. We gratefully acknowledge the computing resources provided on "Fusion," a 320-node computing cluster operated by the Laboratory Computing Resource Center at Argonne National Laboratory.

REFERENCES

[1] "Apache MPM worker documentation." [Online]. Available: http://httpd.apache.org/docs/2.0/mod/worker.html

[2] "Multithreading support in memcached." [Online]. Available: http://code.sixapart.com/svn/memcached/trunk/server/doc/threads.txt

[3] M. Krohn, E. Kohler, and M. Kaashoek, "Events can make sense," in *Proceedings of the USENIX Annual Technical Conference*, 2007.

[4] R. Ashok, *Infiniband Host Channel Adapter Verb Implementer's Guide*, Intel Corporation, March 2003.

[5] Myricom, Inc., "Myrinet Express (MX): A high-performance, low-level, message-passing interface for Myrinet," Version 1.2, 2006.

[6] M. Krishnan, J. Nieplocha, M. Blocksome, and B. Smith, "Evaluation of remote memory access communication on the IBM Blue Gene/P supercomputer," in *Workshop on Parallel Processing at ICPP*, 2008.

[7] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*. MIT Press, Cambridge, MA, 1999.

[8] M. T. Jones, "Boost application performance using asynchronous I/O," IBM developerWorks, 2006.

[9] M. A. Olson, K. Bostic, and M. Seltzer, "Berkeley DB," in *Proceedings of USENIX*, 1999.

[10] I. Red Hat, *Red Hat Database 2.1 Programmer's Guide*, Red Hat, Inc.

[11] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon, *Parallel programming in OpenMP*. San Francisco, CA: Morgan Kaufmann, 2001.

[12] P. Charles et al., "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York: ACM, 2005, pp. 519–538.

[13] W. Carlson, J. Draper, D. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and language specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, 1999.

[14] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, pp. 291–312, August 2007.

[15] E. Allen et al., "The Fortress language specification," Sun Microsystems, Inc., Tech. Rep., 2007. [Online]. Available: http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf

[16] Nvidia Corporation, "CUDA zone." [Online]. Available: http://www.nvidia.com/object/cuda_home.html

[17] Khronos OpenCL Working Group, *The OpenCL Specification, version 1.0.29*, 2008. [Online]. Available: http://khronos.org/registry/cl/specs/opencl-1.0.29.pdf

[18] Apple Inc., "Grand central dispatch." [Online]. Available: http://developer.apple.com/technologies/mac/snowleopard/gcd.html

[19] N. Mathewson and N. Provos, "libevent - an event notification library." [Online]. Available: http://monkey.org/~provos/libevent/

[20] M. Lehmann, "libev." [Online]. Available: http://software.schmorp.de/pkg/libev.html

[21] "The Parallel Virtual File System." [Online]. Available: http://www.pvfs.org

[22] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock, "I/O performance challenges at leadership scale," in *Proceedings of Supercomputing*, November 2009.

[23] T. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, pp. 308–320, 1976.