# Virtual Threads

Elaine Cheong and Fred Reiss
CS262A and CS263
Fall 2000

## Abstract

Multithreaded servers, while relatively simple to design and to implement, tend not to scale well for large numbers of concurrent users. Event-driven servers, which do scale well, are generally more difficult to design, write, and debug than multithreaded servers. Virtual Threads is a new server programming model in which the programmer writes a multithreaded server and a preprocessor automatically converts the server to a sophisticated event-driven server. We describe our current implementation of Virtual Threads and present benchmarks showing that a multithreaded Web server that uses Virtual Threads scales as well as an event-driven server.

# 1 Introduction

Virtual Threads is a new programming model and runtime environment for servers. With Virtual Threads, a programmer writes her program in C as a multithreaded server. The Virtual Threads preprocessor, `vtify`, converts this code to an event-driven model by changing functions to use continuation-passing style. The Virtual Threads runtime environment then executes the converted program as many concurrent state machines.

In Section 1 of this paper, we define terminology that we use in the rest of the paper and explain the motivation behind Virtual Threads. In Section 2, we discuss the design of the Virtual Threads preprocessor and runtime environment. In Section 3, we discuss our current implementation of continuations, the preprocessor, and the runtime environment. We also describe a sample application and preliminary benchmarks of that application. In Section 4, we describes work related to Virtual Threads, and in Section 5, we discuss possible future extensions of Virtual Threads.

## *1.1 Background and Terminology*

In this section, we introduce concurrent server architectures, including multithreaded/multiprocess servers and event-driven servers. Then, we introduce the concept of continuations and continuation-passing style.

### 1.1.1 Concurrent Server Architectures

A *concurrent server* is a server that does work for multiple clients simultaneously. We briefly summarize the dominant design strategies for concurrent servers here. For a more detailed discussion of server design alternatives, we refer the reader to [Stevens1998] and [Pai1999].

#### 1.1.1.1 Multithreaded/Multiprocess Servers

To write a *multiprocess server*, a programmer first writes a program that handles a single connection. Then she writes a second program that runs a copy of the first program to handle each connection that the server receives.

The *multithreaded server* architecture is an optimization of the multiprocess architecture. A multithreaded server spawns a lightweight process, or *thread*, to handle each connection.
We will use the terms *multithreaded* and *multiprocess* interchangeably in this paper.

#### 1.1.1.2 Event-driven Servers

To write an *event-driven server*, a programmer decomposes each transaction between client and server into a series of *events*. Examples of these events include a server receiving a request from a client and a server's disk controller completing a write operation. Having defined a set of events, the programmer then writes a separate module to handle each event.

In a *single-process event-driven* server [Pai1999], a process known as an *event loop* handles all events that occur for all the server's open connections. In such a server, the module that handles an event is a function.

The *asymmetric multiprocess event-driven* server architecture [Pai1999] handles most events with an event loop but uses separate processes when events require blocking operations or heavy computation.

A *pipelined server* is an event-driven server in which the module for each event operates in one or more separate threads of control.

### 1.1.2 Continuations

A *continuation* is an object that stores all the information necessary to perform a future computation. Such an object generally consists of a function and arguments for that function.

The most common definition of continuations defines two operations on them. The `callcc` operation creates a continuation for the current state of the program, while the `throw` operation performs the computation that a continuation represents.

In the *continuation-passing* style of programming, the operations `callcc` and `throw` perform some or all of the program's control flow.

For a more detailed discussion of continuations from the point of view of imperative languages, we refer the interested reader to [Thielecke1999].

## *1.2 Motivation*

Programmers tend to see multiprocess and multithreaded servers as easier to write than event-driven ones. Undergraduate textbooks, such as [Butenhof1997], [Gay2000] and [Stevens1998], teach the multiprocess or multithreaded model first and mention the event-driven model as a specialized approach that is relatively difficult to implement. For example, Butenhof writes, "In most cases [programmers] will find it much easier to write

complex asynchronous code using threads than using traditional asynchronous programming techniques" [Butenhof1997]. Because of its perceived ease of programming, the multithreaded architecture is a popular strategy for implementing servers.

Several researchers have found that multithreaded (and multiprocess) servers do not scale well for the large numbers of concurrent users that today's Internet applications require [Pai1999] [Welsh2000] [Kegel2000]. As the number of concurrent connections to a multithreaded server increases, the overhead of switching between threads and locking shared resources consumes a greater proportion of the server's time. In addition, multithreaded servers have very coarse-grained admission control. Different parts of the server may perform best with different levels of multiprogramming, but, since there is one thread or process per connection, the programming model forces the same level of multiprogramming on all parts of the server.

Event-driven servers largely avoid these performance limitations. Since they operate with a small, fixed number of threads, event-driven servers can avoid most of the task switching and resource locking overhead that plagues multithreaded servers. Additionally, since they handle each portion of a transaction with a separate module, event-driven servers can use the optimal level of multiprogramming for each portion of the server.

However, designing, implementing, and debugging an event-driven server is relatively difficult. The designer of such a server must carefully decompose the server's tasks into a set of events and must assign each of these events to the appropriate functional unit of the server. The implementer must manage the state of each connection separately by hand. Whoever tests and debugs an event-driven server needs to reconstruct the state of each connection by hand, since stack traces only provide information about the server's event loop.

## *1.3 Virtual Threads*

Virtual Threads combine the performance advantages of event-driven server architecture with the simplicity of the multithreaded server architecture. The programmer writes a multithreaded server, and a preprocessor converts this code to an event-driven server. Our preprocessor, called `vtify`, converts parts of each "Virtual Thread" in the original program to continuation-passing style and generates the appropriate continuation objects. These continuations are smaller than the stacks that normal threads would require. Our runtime environment consists of a generic asymmetric multiprocess event-driven server that uses continuations to handle events.

# 2 Design

In this section, we discuss the design of the preprocessor and steps needed to convert a multithreaded program to use Virtual Threads. Then, we describe the design of the runtime environment that executes the Virtual Threads.

## *2.1 Preprocessor*

To convert a multithreaded program to one that uses Virtual Threads, one must convert each thread in the original program to a state machine that stores its state in a continuation. This state machine enters a new state at every point where the original program performed a blocking I/O or called `VT_yield()`. The Virtual Threads preprocessor automatically performs the conversion from a multithreaded C program to a state machine and also generates the appropriate continuations.

In the following sections, we first introduce the concept of multistate functions and the structure of a Virtual Thread continuation. Then, we describe the stages of preprocessing that our preprocessor uses to convert a multithreaded program to a Virtual Threads program.

### 2.1.1 Multistate Functions

To make preprocessing more efficient, we divide the functions in a multithreaded program into two classes: *single-state functions* and *multistate functions*. As its name suggests, a multistate function is a function that corresponds to more than one state of the state machine for a Virtual Thread, whereas a single-state function executes entirely inside one state. More precisely, we define multistate functions by the following two rules:

1. The functions `VT_yield()`, `VT_read()`, and `VT_write()` are multistate functions.
2. Any function that calls a multistate function is also a multistate function.

### 2.1.2 Virtual Threads Continuations

A Virtual Thread continuation contains all of the information that a Virtual Thread will need for any future computation. Figure 1 shows the continuation for

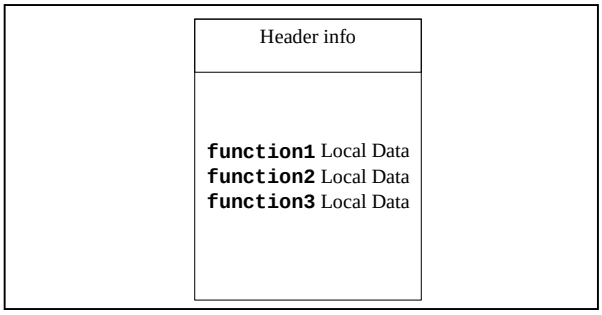


**Figure 1: Structure of a Virtual Thread continuation.**

a Virtual Thread that uses three multistate functions: `function1`, `function2`, and `function3`. The continuation contains local variables for each multistate function that the Virtual Thread calls, along with a header, which stores tracking information for the Virtual Thread.

The Virtual Threads preprocessor rewrites multistate functions so that the functions store some or all of their variables inside a continuation instead of on the stack. These preprocessor conversions guarantee that a complete continuation is always present when the state machine for a Virtual Thread reaches a state boundary. In effect, the preprocessor blends the *callcc* operation into the execution of each multistate function. The preprocessor generates a single "outer" function for each state machine, and the runtime environment places a pointer to this function into the header for each continuation. *Throwing* to a Virtual Threads continuation consists of calling this function pointer and passing the function a pointer to the continuation.

In order to prevent continuations from being of unbounded size, we do not allow recursion among multistate functions. Also, we only allow the Virtual Threads runtime environment to throw to a continuation once. This second constraint means that the preprocessor is free to generate code that reuses old continuations with updated local data, instead of constructing new ones when state transitions occur.

It is highly probable that, when the Virtual Threads runtime environment invokes a continuation, the continuation will not be in the computer's data cache. In order to minimize the cost of bringing continuations into the cache, it is important to make the continuations as small as possible. The preprocessor minimizes continuation size by applying two principles:

1. If the value of a local variable does not need to be saved across a state boundary, then that variable does not need to occupy space in the continuation. Instead, the variable can reside on the event loop's stack, which is generally already in the processor's cache. In particular, if a function is not multistate, then none of its variables need to be stored in the continuation.
2. If two local variables (not necessarily in the same function) do not contain live data at the same time, then those variables may occupy the same space in the continuation.

In general, the properly-optimized continuation for a Virtual Thread will require less space than the stack for the equivalent conventional thread.

## 2.1.3 Stages of Preprocessing

The preprocessor makes three passes through the code for each thread in a multithreaded program. During the first pass, the preprocessor identifies which functions are multistate. The second pass generates a continuation for each multistate function and rewrites the multistate functions so that they store variables in the continuations. During the third pass, the preprocessor merges the individual continuations for the multistate functions into a single continuation and merges all of the multistate functions into a single function.

### 2.1.3.1 Identifying multistate functions

In its first stage, the preprocessor identifies which functions contain multiple states. The preprocessor makes this judgment in one pass using the following algorithm:

1. Construct a call graph for all functions in the original program.
2. Reverse the direction of each edge in the call graph such that each edge points from *callee* to *caller*.
3. Starting from the multistate functions `VT_read()`, `VT_write()`, and `VT_yield()`, perform a breadth-first search of the reversed call graph, marking as multistate each function that calls a multistate function.

If the preprocessor encounters a cycle among multistate functions while performing step 3 of the above algorithm, then the original program contains recursive multistate functions and is not valid input for the preprocessor.

### 2.1.3.2 Converting multistate functions

In the second stage, the Virtual Threads preprocessor makes all of the necessary conversions on the multistate functions: (1) generating preliminary continuations, (2) converting multistate functions to use these preliminary continuations, and (3) inserting state boundary labels.

#### *2.1.3.2.1 Generating preliminary continuations*

In the first step of the conversion stage, the preprocessor generates a preliminary continuation for each multistate function. A preliminary continuation contains only local data for its associated function. Figure 2 shows a sample multistate function (`function1`) called by a thread in an imaginary multithreaded program. In order to generate a continuation for this function, the preprocessor must first locate all of the local data that the function uses. This data includes function parameters (`arg1`) and

```
int function1(int arg1) {
    int local1, local2;
    local1 = function2(arg1);
        /* function2 is multistate. */
    local2 = function2(arg1 + local1);
    return local1 + local2;
}
```

**Figure 2: Sample multistate function.**

local variables (`local1` and `local2`). The preprocessor then creates a preliminary continuation, allocating space for the each local variable the value of which needs to remain in memory across a state boundary. Figure 3 shows a preliminary continuation for `function1`. Note that the variable `local2` does not need to reside in the continuation, because `function1` only accesses `local2` in a single state.

#### *2.1.3.3.2 Converting functions to use preliminary continuations*

In the second step of the conversion stage, the preprocessor rewrites all of the multistate functions so that they use the generated preliminary continuations. To do this conversion, the preprocessor first replaces the function parameter list with a pointer to the preliminary continuation object. Then, in the body of the function, the preprocessor replaces references to local data with references to data now stored as members of the preliminary continuation object.

#### *2.1.3.3.3 Inserting state boundaries*

In the third step of the conversion stage, the preprocessor inserts labels that mark state boundaries in the multistate functions. Recall that state boundaries occur at places where a Virtual Thread performs blocking I/O or calls `VT_yield()`.

### 2.1.3.3 Merging multistate functions

During its final stage, the preprocessor merges the multistate functions and their continuations to form a Virtual Thread. The preprocessor performs this merging in three steps: (1) generating a Virtual Thread continuation as described in Section 2.1.2, (2) converting the multistate functions to continuation passing style, and (3) merging the multistate functions into a single function.

#### *2.1.3.3.1 Generating the Virtual Thread continuation*

In the first step of the merge stage, the preprocessor generates a Virtual Thread continuation by merging all of the preliminary continuations that it created in the previous stage. The preprocessor minimizes the size of the merged continuation by following the principles outlined in Section 2.1.2. The preprocessor adds a

```
struct _VT_cont_function1 {
    int arg1;
    int local1;
}
```

**Figure 3: Preliminary continuation.**

header to the beginning of the merged continuation. The header includes a function pointer, which points to the Virtual Thread function generated in Section 2.1.3.3.3, and a state ID, which records the next state to be executed.

#### *2.1.3.3.2 Converting multistate functions to continuation passing style*

In the second step of the merge stage, the preprocessor converts all of the multistate functions to use the merged continuation instead of the preliminary continuations used in the previous stage. The preprocessor performs this conversion by changing calls to multistate functions to *continuation-passing style*, in which the functions use the Virtual Thread continuation and a single shared stack frame in place of multiple frames on the C runtime stack. The preprocessor generates temporary variables to hold function arguments and return values and rewrites the multistate functions to use these temporary variables.

#### *2.1.3.3.3 Creating the Virtual Thread*

In the last step of the merge stage, the preprocessor takes the bodies of the multistate functions and merges these *subfunctions* into one Virtual Thread function.

In order for the Virtual Thread function to jump to the appropriate state when the runtime environment passes it a continuation, the preprocessor inserts a *jump table* at the beginning of the function. This jump table contains entries for each state boundary that the preprocessor inserted in the previous stage of preprocessing, in addition to a special “start” state.

Since all of the subfunctions use the continuation to store their local data, a Virtual Thread can stop at any state boundary and have a continuation already prepared. The runtime environment can then *throw* to another Virtual Thread’s continuation, until the other Virtual Thread reaches a state boundary. Once this second thread is done executing, the runtime environment can then restore the first Virtual Thread by *throwing* to its continuation.

In the next section, we describe the design of the runtime environment that controls the execution of a Virtual Thread.

## *2.2 Runtime Environment*

The runtime environment for Virtual Threads is modeled after the asymmetric multiprocess event-

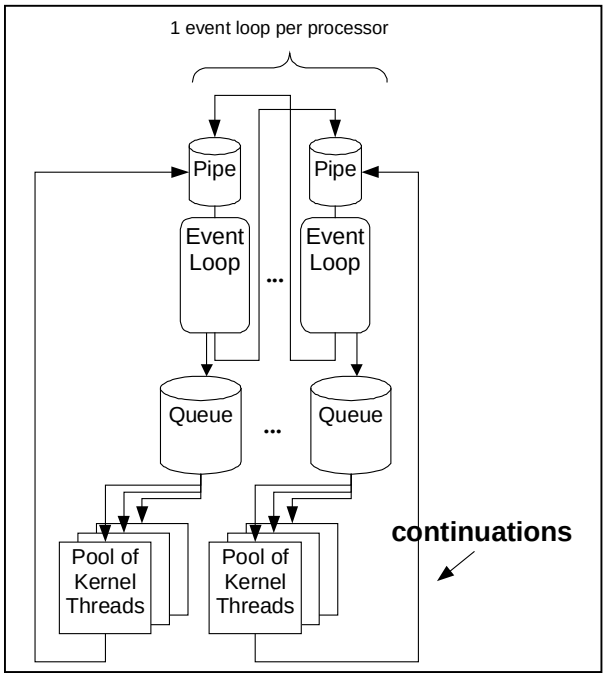


**Figure 4: Virtual Threads runtime environment.**

driven architecture described in [Pai1999]. We extend this architecture to include multiple event loops, and we replace the architecture's external processes with kernel threads so that the entire server runs inside a single address space. Figure 4 illustrates the architecture of the Virtual Threads runtime environment.

## 2.2.1 Event Loops

The Virtual Threads runtime environment creates an event loop for each processor of the computer. Each event loop runs in a separate kernel thread and maintains a separate mapping from events to continuations. When an event occurs, the event loop throws to the appropriate continuation. A continuation belongs to no more than one event loop at a time.

Each event loop maintains a pair of *throwable queues*, which contain continuations for runnable Virtual Threads. A throwable queue is similar to the runnable queue of a conventional thread scheduler. At any given time, one of these queues is the *active* queue and the other is the *yield* queue. When a Virtual Thread calls `VT_yield()`, the Virtual Thread inserts its own continuation into the yield queue. The algorithm for the event loop is shown in Figure 5.

### 2.2.1.1 Metathreads

Each event loop has a pipe (See `pipe(2)`.) that it uses as a communication mechanism. A Virtual Thread may "migrate" to another event loop by placing a pointer to its continuation in the other event loop's pipe. Each event loop has a special Virtual Thread, called the

```
Repeat forever:
    For Each event e that has occurred
        Add the continuation associated with event e
            to the active queue.
    End For Each

    For Each continuation c in the active queue
        Throw to c.
    End For Each

    Swap the active and yield queues.
End Repeat
```

**Figure 5: Algorithm for a Virtual Threads event loop.**

*Metathread*, which reads continuations from this pipe and places them on the yield queue.

## 2.2.2 Thread Pools

Some operations do not lend themselves to running in a single-threaded event loop. For example, the programmer may need to use APIs that block the calling kernel thread, or she may need to write CPU-intensive functions that are difficult to schedule both nonpreemptively and fairly. To deal with these situations, each Virtual Threads event loop maintains a pool of "worker" kernel threads. The worker threads in each pool read continuations from a queue and *throw* to them. The Virtual Threads library provides functions that allow the programmer to specify that sections of her code run on a worker thread. While running on a worker thread, a Virtual Thread can perform blocking operations, use POSIX thread locks, create new kernel threads, and perform computations of arbitrary length. When a Virtual Thread finishes executing on a worker kernel thread, the worker thread reinserts the Virtual Thread into the event loop by writing a pointer to its continuation into the event loop's pipe.

## 2.2.3 Clues

Having multiple event loops creates a need to distribute Virtual Threads among the event loops. One approach to allocating Virtual Threads to event loops is to use a centralized scheduler, placing each new Virtual Thread in the event loop with the lowest load. Unfortunately, this centralized approach would lead to resource contention and poor cache locality. Since each Virtual Thread would execute entirely in a given event loop, event loops would contend for global resources. Furthermore, since each event loop would execute every line of a given Virtual Thread, the working set size of an event loop could easily exceed the size of the processor cache by a large margin [Larus2000].

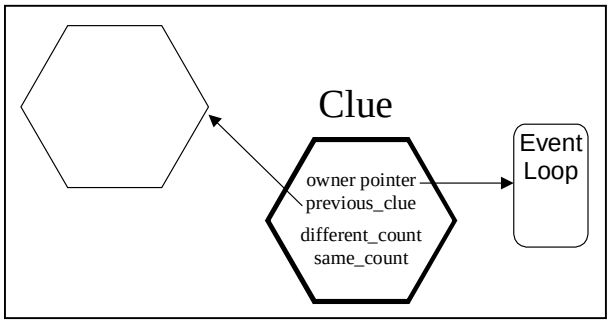


**Figure 6: Clues create self-organizing pipelined servers.**

To avoid these potential bottlenecks, the Virtual Threads runtime environment uses a distributed scheduler based on *clues.* A *clue*[1] (See Figure 6) is an object that belongs to exactly one event loop. A Virtual Thread may *follow* a clue by migrating to the event loop that owns the clue. If the Virtual Thread is already at the appropriate event loop, it does nothing. Clues provide a mechanism for creating self-organizing pipelined servers. The programmer divides her code into pipeline stages by following a clue at the beginning of each section of code that corresponds to a stage. At runtime, each stage of the programmer's original code runs in exactly one event loop. If there are more stages than processors, multiple stages run in the same event loop.

### 2.2.3.1 The Cluethread

When the Virtual Threads runtime environment starts, it arbitrarily assigns clues among event loops. To balance the load between event loops, a special Virtual Thread called the *Cluethread* periodically moves clues from one event loop to another to ensure that each event loop, and hence each processor, does the same amount of work.

The Cluethread spends most of its time sleeping inside a kernel thread pool. Periodically, the Cluethread wakes up, reads the load of each event loop, and reallocates clues if necessary. Each event loop in the Virtual Threads runtime environment keeps track of its own processor usage[2]. To avoid race conditions, the Cluethread migrates to a given event loop before checking the event loop's processor usage or moving any of the event loop's clues.

The Cluethread uses a simple heuristic to decrease the communication between event loops. The runtime environment keeps track of clues that Virtual Threads always follow in sequence[3]. When the Cluethread needs to move a clue to balance the load, it attempts to move the clue that has the minimal number of following or preceding clues on the same event loop.

# 3 Current Implementation

This section describes our current implementation of Virtual Threads. First, we describe how we have implemented continuations in C using structs and how our current implementation optimizes the size of these structs. Second, we describe the current status of the preprocessor and the actual steps that are needed to convert a multithreaded program to use Virtual Threads. Finally, we describe our current implementation of the Virtual Threads runtime environment and present a sample application and some benchmarks of that application.

## *3.1 Continuations*

As described in Section 2.1.2, a Virtual Thread continuation contains local data for each multistate function that a thread calls in a multithreaded program.

We will use an example to explain our current implementation of Virtual Thread continuations. In this example, a main server thread listens for connections and spawns a child thread to handle each connection. Each child thread runs the function `echo_thread` to handle its connection. Figure 7 shows the C code for this function and the Virtual Threads continuation for the thread. Let us examine the structure of the continuation from bottom to top.

The bottom of the continuation contains the local data for `echo_thread`, including local variables, function arguments, return value, and return state. We wrap the function arguments with an alternate generic name for each variable inside of a union so that we can refer to the arguments with generic macros later.

[1] According to the *Merriam-Webster Collegiate Dictionary*, a clue is "something that guides through an intricate procedure or maze of difficulties."

[2] Under Linux, the `clock(2)` system call returns the amount of processor time that the calling kernel thread has used.

[3] To help the Cluethread keep track of clues that Virtual Threads follow in sequence, each clue has three fields: `previous_clue`, `different_count`, and `same_count` (See Figure 6). `previous_clue` is a pointer to another clue object, while the other two fields are integers. When a Virtual Thread follows a clue, the Thread compares the clue's `previous_clue` field to the last clue that the Thread followed. If the two clues are the same, then the Virtual Thread increments the `same_count` field. Otherwise, the Virtual Thread sets `previous_clue` to point to the last clue that the Thread followed and increments `different_count`.

```
int echo_thread( int sock ) {
    char c;
    while( _VT_ioReadSock( sock, &c, 1 )
        == 1 ) {
        _VT_ioWriteSock( sock, &c, 1 );
    }
    return 0;
}

/* Continuation for echo_thread */
①struct _VT_cont_echo_thread {
    union {
        /* The header. */
        _VT_gdataConthdr_t _VT_header;

        /* Continuations for
            multistate functions that
            echo_thread calls */
      ② struct _VT_cont__VT_ioReadSock
            _VT_child__VT_ioReadSock;
      ③ struct _VT_cont__VT_ioWriteSock
            _VT_child__VT_ioWriteSock;
    };

    /* Where echo_thread returns to. */
    _VT_state_t _VT_retstate;

    /* What is returned. */
    int _VT_retvalue;

    /* Function arguments */
    union {
        int _VT_arg1;    /* AKA sock. */
        int sock;
    };

    /* Local variables in function body. */
    char c;

};
```

**Figure 7: Sample thread with Virtual Threads continuation.**

The continuation for each multistate function called by `echo_thread` appears above the local data. In this example, `conn_thread` calls two multistate functions: `_VT_ioWriteSock`, and `_VT_ioReadSock`. These functions are built-in Virtual Threads wrappers for blocking I/O system calls. The continuation for each of these functions contains their corresponding local data.

A header, which appears at the top of the continuation, contains tracking information for the Virtual Thread. Note how, because of a union, the header appears at the beginning of every multistate function's continuation. Our current implementation of Virtual Threads continuations makes merging preliminary continuations very simple.

Our current implementation of continuations optimizes for space at the level of granularity of individual functions. Since `echo_thread` calls its two functions sequentially, it does not need to save the states for both functions at the same time. We use a union to share the local data space for all of the functions that `echo_thread` calls. The next few

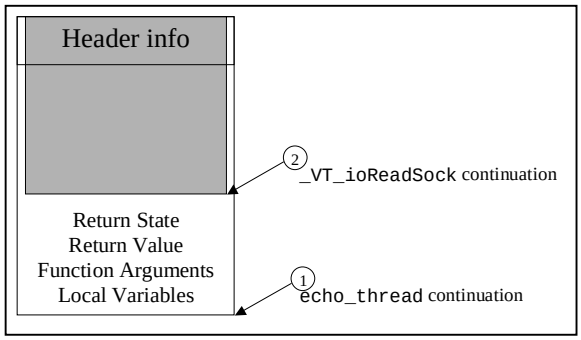


**Figure 8: echo_thread continuation during call to _VT_ioReadSock.**

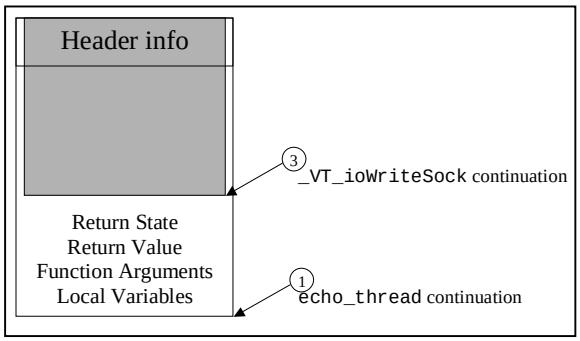


**Figure 9: echo_thread continuation during call to _VT_ioWriteSock.**

paragraphs will help to explain these optimizations in further detail.

Figure 8 illustrates the structure of the `echo_thread` continuation ① at a state boundary where the Virtual Thread is inside the `_VT_ioReadSock` function. At this point, the Virtual Thread stores the local data for `_VT_ioReadSock` within a continuation ② inside of the `echo_thread` continuation.

Figure 9 illustrates the structure of the `echo_thread` continuation ① at a state boundary where the Virtual Thread is inside the `_VT_ioWriteSock` function. Note that the Virtual Thread has stored the `_VT_ioWriteSock` continuation ③ inside of the `echo_thread` continuation.

## 3.2 Preprocessor

Our preprocessor, `vtify`, uses David Gay's C parser to generate an abstract syntax tree (AST) from arbitrary Gnu C code. First, `vtify` searches through the top level of the AST for function declarations and generates preliminary continuations (as described in Section 2.1.3.2.1). Next, `vtify` replaces the arguments in the function parameter list with a pointer to the Virtual Thread continuation. Then, it removes all local variable declarations from the function body. If the function declares and initializes any of the local variables in the same statement, `vtify` replaces this statement with an

assignment to the local variable. Finally, `vtify` replaces all references to the original local variables with references to members of the preliminary continuation.

Currently, the programmer must merge the preliminary continuations together by hand. Due to the structure of our current implementation of continuations, this operation is quite mechanical. The programmer only needs to insert lines for the function's header, return state, and return value into the continuation; and to place at the top of the continuation instances of the continuations for the other multistate functions that the function calls.

The current implementation of Virtual Threads supplies a library of C preprocessor macros to convert calls to multistate functions to continuation-passing style. The programmer needs to replace these types of function calls with the Virtual Threads macros. This replacement is a simple cut-and-paste operation.

Finally, the programmer copies and pastes the bodies of the multistate functions into a single function and inserts a label at the beginning of each function. Using another C preprocessor macro that our library provides, the programmer creates a switch statement at the beginning of the top-level function to act as a dispatch table.

To summarize, our current implementation requires that the programmer perform some operations by hand. However, these operations are mechanical and could easily be automated by extending the `vtify` preprocessor.

## *3.3 Runtime Environment*

We have implemented most of the multiprocessor version of the Virtual Threads runtime environment, including clues, Metathreads, worker thread pools, multiple event loops in separate kernel threads, thread-safe queues, and thread-safe memory pools for allocating continuations efficiently. However, since we do not have easy access to a multiprocessor, we have only tested and debugged our runtime environment in a single-processor configuration, with a single event loop and one pool of worker threads.

## *3.4 Sample Application and Preliminary Benchmarks*

As a test of our implementation of Virtual Threads, we converted Peter Sandvik's Simple Web Server [Sandvik2000] from a multiprocess architecture to a multithreaded architecture using POSIX threads. We then converted this multithreaded server to Virtual Threads. With our current preprocessor and libraries, the conversion from POSIX threads to Virtual Threads required approximately two and a half hours, whereas the conversion from forked processes to POSIX threads required approximately eight hours.

To test the performance of our current implementation, we ran the WebStone benchmark [Mindcraft2000] on the Simple Web Server, converted to POSIX threads and to Virtual Threads. We also ran the benchmark on the Apache Web server for comparison purposes.

### 3.4.1 Benchmark Setup

The server that we used for benchmarking was a personal computer running Redhat Linux 7.0 on a 400 MHz Pentium II processor with 128 MB of memory. The client was an IBM Thinkpad T20, also running Redhat Linux 7.0 on a 700 MHz Pentium III processor with 128 MB of memory. We connected both computers to a 100base-TX Ethernet hub. There were no other computers attached to the hub.

Both machines ran in multiuser mode, but we disabled their X servers and quiesced the machines before running benchmarks. Before each run of the benchmark, we restarted the server process on the server machine and ran a 2-minute "dry run" of the benchmark to put the server in a consistent state. Each run of the benchmark lasted five minutes.

WebStone's default workload consists of ten files of varying sizes. Since all ten files fit easily into the server machine's file system buffer cache, the default workload produced very uninteresting results. In particular, all the servers ran so quickly that the server's network card was the only factor limiting throughput, and HTTP transactions finished so quickly that the WebStone clients had difficulty keeping more than 70 connections to the server open at a time.

To remedy these problems, we created an alternative workload consisting of 250 1-megabyte files. This workload did not fit in the server's buffer cache. The architecture of the WebStone clients prevented us from using a larger number of smaller files.

We tested each server with numbers of client connections ranging from 1 to 500.

### 3.4.2 Benchmark Results

The results of our benchmarks are shown in Figure 10. The Virtual Threads Web server produced no errors, even when serving 500 clients simultaneously. When serving more than 2 clients, the Virtual Threads Web server outperformed both the other servers by as much as 250%. These results are similar to those reported in [Welsh2000] and [Pai1999], in which event-driven servers scaled better than multithreaded servers on I/O-intensive workloads.

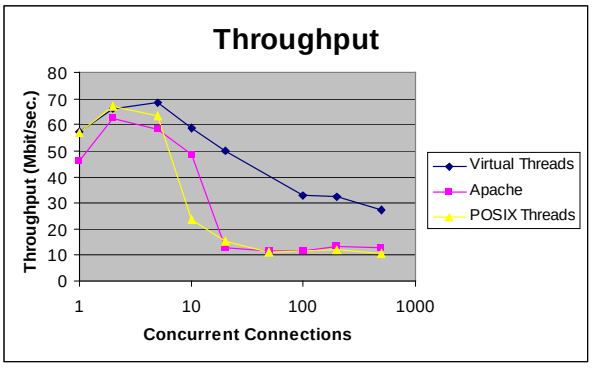


**Figure 10: Benchmark results.**

# 4 Relation to Previous Work

Thread libraries are a very well-studied research area, and a full discussion of past research on threads is beyond the scope of this paper. [Kavi1998] provides a survey of existing implementations of multithreading.

Two recent user-level thread libraries that are similar to Virtual Threads are Gnu Portable Threads [Engelschall2000] and Silicon Graphics State Threads [SGI2000]. Like Virtual Threads, these libraries schedule their threads using an event loop. However, the libraries keep a separate stack and register file for each thread and must bring the entire stack and register file into the cache on a context switch. The thread libraries do not have any support for multiple processors, and their internal implementations would require a complete rewrite to use multiple event loops or clues. Furthermore, these libraries force the programmer to use only nonpreemptive thread scheduling, whereas Virtual Threads allow the programmer to use thread pools when nonpreemptive scheduling is inconvenient.

The literature on continuations is almost as large as the literature on threads. We refer the systems-minded reader to [Thielecke1999] for an accessible introduction to the concept of a continuation.

Functional languages have used continuations to implement threads for the past twenty years [Wand1980]. Safe functional languages make continuations easier to implement than context switches.

Some researchers have used continuations to implement threads in imperative languages. For example, Python Microthreads [Tismer2000] is a thread package that uses the continuation support of Stackless Python [Tismer1999]. In unsafe imperative languages such as C, however, continuations are considerably more difficult to implement, and there has been little work on thread libraries that use them. The user-level threads package for the Mach operating system [Dean1993] used continuations in a limited context inside the library functions. Virtual Threads are different from other implementations of threads in C in that our preprocessor rewrites programs to use continuations explicitly.

# 5 Future Work

The work we have performed this semester has produced promising early results but leaves room for additional innovation.

## *5.1 Preprocessor*

Our current implementation of the Virtual Threads preprocessor requires that the programmer perform some parts of the conversion to a state machine by hand. The steps that currently require human intervention are very mechanical in nature, and the preprocessor could easily perform those steps automatically. Future versions of the preprocessor should be entirely automatic, taking C source code as their input and calling the C compiler to generate object files as their output.

Future generations of the Virtual Threads preprocessor should also perform more aggressive optimizations to reduce continuation size. The preprocessor should use a dataflow analysis to determine which variables a Virtual Thread needs to save across each state boundary and should rewrite multistate functions to store only those variables in continuations, and only at those points in the Virtual Thread where the next time that the Thread reads from the variables could be during a different state. All other variables can reside in registers or on the event loop's stack, which is more likely to be in the processor's cache than is the continuation.

Due to the difficulty of performing alias analysis, it may be difficult for the preprocessor to determine whether a buffer that is a local variable in a Virtual Thread is "live" at a given point in the program. However, the Virtual Threads runtime environment already uses memory pools to allocate Virtual Threads continuations. A simple work-around for the alias analysis problem would be to allow the programmer to use these memory pools to allocate buffers dynamically.

To make optimal use of the current implementation of Virtual Threads, the programmer needs to know where in her code to insert calls to `VT_yield()`. Future implementations of the Virtual Threads preprocessor should free the programmer of this requirement by inserting these yield calls automatically. The fact that it is sometimes impossible to predict statically how long a given section of code will take to execute would make it difficult for the preprocessor to insert yields into parts of some programs. To deal with those cases, the

preprocessor could direct sections of the program to use a thread pool instead of running in an event loop.

The current implementation of Virtual Threads has only limited support for dynamic libraries. In particular, a dynamic library function must execute within a single state of a Virtual Thread, so a dynamic library function that blocks or performs complex computations needs to run inside a worker thread pool. A simple mechanism for allowing multistate functions inside dynamically-loaded libraries is to suspend the calling Virtual Thread, spawn a second Virtual Thread inside the library to run the function, and return control to the first Virtual Thread when the second Virtual Thread completes. Future versions of the Virtual Threads preprocessor should automate the process of converting multistate functions in dynamic libraries.

## 5.2 Runtime Environment

We have not yet used Virtual Threads on a machine with more than one processor. An important piece of future work is to obtain a multiprocessor machine in order to test and to debug the Virtual Threads runtime environment with multiple event loops running in parallel. Such an environment would allow us to determine the effectiveness of clues for creating self-tuning pipelined servers.

The current implementation of Virtual Threads uses worker thread pools of a fixed size. However, past research has demonstrated that different tasks perform better at different levels of multiprogramming. It would be beneficial, therefore, for Virtual Threads to have a mechanism of dynamically adjusting the size of the worker thread pools to obtain the highest possible level of throughput.

The concept of clues offers several intriguing future directions. On computers with a large number of processors, the programmer may wish for two or more event loops to share a clue for increased parallelism. One mechanism to provide this sharing is a *hashed n-clue*. A hashed *n*-clue consists of *n* "subclues", numbered 1 to *n*, and a hash function. To follow a hashed *n*-clue, a Virtual Thread applies the hash function to its continuation pointer to obtain an index from 1 to *n* and follows the appropriate subclue. Multiple hashed *n*-clues could create pipelined servers with superscalar pipelines.

# 6 Conclusion

We have successfully implemented an early version of Virtual Threads, a system for converting simple multithreaded servers to scalable event-driven servers. Our implementation consists of a preprocessor that converts threads written in C into state machines that store information in continuations and a generic asymmetric multiprocess event-driven server to run those state machines. Preliminary benchmarks of our implementation applied to a simple Web server show that Virtual Threads provide the performance benefits of an event-driven server while allowing the programmer to use the intuitive multithreaded server programming model.

# 7 References


[Butenhof1997] Butenhof, David R. *Programming with POSIX Threads*. Addison Wesley, 1997.

[Dean1993] Dean, Randall. "Using Continuations to Build a User-Level Threads Library." Third USENIX Mach Conference, April 1993.

[Engelschall2000] Engelschall, Ralf. *Gnu Portable Threads*. Software Library. First released February 2000. http://www.gnu.org/software/pth/.

[Gay2000] Gay, Warren W. *Linux Socket Programming by Example*. Que, 2000.

[Kavi1998] Kavil, Krishna. "Multithreading Implementations." *Microcomputer Applications*, vol. 17, no. 3, 1998.

[Kegel2000] Kegel, Dan. "The C10K Problem." http://www.kegel.com/c10k.html.

[Larus2000] Larus, James and Michael Parkes. "Enhanced Server Performance with StagedServer." Lecture. U.C. Berkeley, October 2000.

[Mindcraft2000] Mindcraft Inc. *WebStone 2.5b3*. Computer program. http://www.mindcraft.com/webstone.

[Pai1999] Pai, V. S. et al. "Flash: An Efficient and Portable Web Server." USENIX 1999.

[Sandvik2000] Sandvik, Peter. *Simple Web Server*. Computer program. http://linuxstuffs.cjb.net.

[SGI2000] Silicon Graphics, Inc. *State Threads*. Software library. First released June 2000. http://oss.sgi.com/projects/state-threads/.

[Stevens1998] Stevens, W. Richard. *UNIX Networking Programming Volume 1—Networking APIs: Sockets and XTI*. Second ed. Prentice Hall PTR, 1998.

[Thielecke1999] Thielecke, Hayo. "Continuations, Functions and Jumps." *SIGACT News*, vol. 30, no. 2, June 1999.

[Tismer1999] Tismer, Christian. *Stackless Python*. Software library. First released 1999. http://www.stackless.com/.

[Tismer2000] Tismer, Christian et al. *Python Microthreads*. Software library. http://world.std.com/~wware/uthread.html.

[Welsh2000] Welsh, Matt et al. "A Design Framework for Highly Concurrent Systems." Submitted for publication, April 2000.