S: a Scripting Language for High-Performance RESTful Web Services

Daniele Bonetta Achille Peternier Cesare Pautasso Walter Binder

Faculty of Informatics, University of Lugano – USI
Lugano, Switzerland
{name.surname}@usi.ch

Abstract

There is an urgent need for novel programming abstractions to leverage the parallelism in modern multicore machines. We introduce *S*, a new domain-specific language targeting the server-side scripting of high-performance RESTful Web services. *S* promotes an innovative programming model based on explicit (control-flow) and implicit (process-level) parallelism control, allowing the service developer to specify which portions of the control-flow should be executed in parallel. For each service, the choice of the best level of parallelism is left to the runtime system. We assess performance and scalability by implementing two non-trivial composite Web services in *S*. Experiments show that *S*-based Web services can handle thousands of concurrent client requests on a modern multicore machine.

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

General Terms Languages, Performance

Keywords RESTful Web services, Multicores, Scalable service execution

1. Introduction

Even if modern server infrastructures for hosting Web services feature highly parallel multicore processors, most existing software and programming languages make it a challenge to fully benefit from the power of such hardware platforms. Given the intrinsically parallel nature of service-based applications, the opportunity exists to design service-oriented architectures which can take advantage of the asynchronous, message-based interactions between independent (i.e., share-nothing) services to make efficient usage of multicore hardware.

In this paper we describe the design of a new language, called *S*, which targets the domain of RESTful Web service [10] development and composition. We find that stateless interactions among RESTful Web services and the explicit management of their state is very useful to identify which parts of a service-oriented architecture can be parallelized.

Even if it is possible to design scalable systems using any language, we believe that embedding specific constraints in the design of a language can make the corresponding guidance more directly available to developers. Thus, we also discuss our novel implicit state-oriented programming model, which has been embedded in the design of the *S* language.

S features native support for architectural-level abstractions, such as services and resources, and allows developers to script service behavior in terms of request handlers associated with the uniform interface of each resource. The state of resources is explicitly marked so that the compiler can statically distinguish the functional behavior from the stateful elements and provide the runtime with enough information such that the correct parallelization strategy can be applied. Likewise, developers do not have to worry about synchronization issues due to concurrent client requests as these are dealt with by the runtime system. The *S* compiler currently includes a backend targeting the JavaScript language and makes use of a runtime based on Node.js [19]. It features automatic code de-synchronization (as Node.js is a single-process, asynchronous server) as well as out-of-order parallel execution. As we demonstrate with non-trivial case studies, the *S* runtime can transparently and efficiently parallelize the execution of RESTful Web services across multiple CPU cores while serving thousands of concurrent clients.

This paper makes the following contributions: it introduces the design and the parallel programming model of the *S* language. It describes its compiler, which detects which portions of code can be executed in parallel and automatically optimizes them for concurrent execution. It presents the main architectural patterns used at runtime to parallelize the execution of the *S* language featuring dynamic replication of services and automatic load balancing.

The rest of this paper is structured as follows: in Section 2 we provide the necessary background to understand the characteristics of the HTTP protocol that we leverage in the design of the *S* language, as described in Section 3. Section 4 presents the *S* language syntax. In Section 5 we explain how the language is compiled into server-side JavaScript to be executed on the *S* runtime, described in Section 6. Sections 7 and 8 provide an evaluation of the *S* runtime system. Section 9 presents related work, and Section 10 concludes this paper.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

2. Background

2.1 RESTful Web Services and the HTTP Protocol

RESTful services are Web services which make full usage of the HTTP protocol [10]. HTTP is the client/server protocol at the core of the World Wide Web [1]. It is based on the notions of globally addressable *resource* (any element published by the server that the client can interact with), stateless interactions (no state should be shared between clients and the server after a request has completed so that servers can treat every request independently from the previous ones), and uniform interfaces (the fixed and well-defined set of possible "actions" that the client can perform on any resource). The uniform interface of HTTP consists of the so-called HTTP methods. The main methods are GET, PUT, DELETE, and POST, which are used to read, create/modify, delete, or access resources. A key aspect of the HTTP protocol is that any method has a precise effect on the state of the receiving resource. GET requests, for instance, are not expected to alter the state of the resource they are applied to. Therefore, multiple GET requests can be safely processed in parallel. Likewise, the value (or representation) of resources fetched using GET methods can be cached. PUT and DELETE methods, on the other hand, are *idempotent* methods: no matter how many times the same PUT/DELETE request will be issued on the same resource, the result will be identical. As a consequence, such requests can be processed by the server in a non-deterministic order and can be retried in case of failure as many times as needed. Conversely, POST methods do not have any property, thus, POST requests must be executed exactly once. Resources can also be retrieved using different representation formats (e.g., HTML, plain text, PDF, JSON, XML). We do not further discuss this aspect as it does not affect parallelism, which is the main focus of this paper.

2.2 JavaScript for Server-Side Development

JavaScript is the most widely used language for client-side development of Web applications. Its distinguishing features are its flexibility, its prototype-based object orientation, its functional nature, as well as its HTML interoperability based on the Document-Object Model (DOM). Despite of being perceived as an inefficient language (like many other dynamically typed languages), JavaScript code is nowadays executed very efficiently. In fact, its wide distribution has spawned many engineering efforts directed at the implementation of high-performance JavaScript virtual machines, such as Google's V8, Safari's Webkit, and Firefox's SpiderMonkey. Given its origins as a scripting language embedded in the Web Browser, JavaScript has not been originally designed to express process-level parallelism. It has only been recently extended to offer support for parallel computations structured according to the master/worker pattern with HTML5 WebWorkers [13].

The increasingly good performance offered by JavaScript virtual machines has motivated the adoption of JavaScript also as a server-side scripting language. Notable in this field is Node.js: a server-side JavaScript framework running on top of Google's V8 which can be used for the development of I/O-bound networking-intensive applications. Node.js features an asynchronous event-driven runtime based on a single-process event-based architecture inspired by the one of Python's Twisted [9], which enables it to handle thousands of concurrent requests on a single V8 instance. Given its single-threaded design (which helps to deal with many concurrency issues), Node.js can exploit modern multicore machines only with approaches based on the master/worker pattern. Also, its asynchronous, event-driven programming model could result to be verbose in complex service development. Conversely, having JavaScript on the server-side clearly represents a great advantage for the end-to-end development of service-oriented applications. Thus, we have chosen to base the design of the *S* language on JavaScript and add the missing features (i.e., parallelism and modularization in terms of services and resources), as described in the following sections.

3. The Design of the *S* Language

S is an extension of the JavaScript programming language targeting the design of service-oriented architectures, with particular focus on RESTful Web service development and composition. *S* extends JavaScript with new features for service scripting, such as synchronous interaction primitives, out-of-order parallel execution of I/O-bound tasks, as well as declarative support for publishing, consuming, and composing REST resources. The aim of the language is to enable the development of high-performance RESTful Web services. The main design drivers are as follows:

High Abstraction Level. The language introduces novel primitives such as *services*, *resources*, and *request handlers* to the JavaScript language. These abstractions address the lack of modularity constructs in JavaScript and provide scoping and lifecycle semantics specific to RESTful Web services. The goal is to let the developer declare the structural decomposition of a service-oriented architecture so that the compiler and the runtime have enough information to derive which services and which resources can be replicated for scalability and parallelization purposes.

Simple Parallel Programming Model. Defining a programming model to let the developer exploit the parallelism available in modern hardware represents a major challenge. *S* embraces a simple yet powerful programming model with the main aim of easing the parallelization of services. In this way, the developer can focus on the semantics of the interaction among different services, and delegate the parallelism management mostly to the runtime.

JavaScript Support. The language is designed to be JavaScript compatible. This has the notable advantage of bringing all the features of a client-side language (for instance, the DOM and JSON support) to a server-side language, without sacrificing performance (thanks to the Node.js-based runtime).

3.1 The Programming Model of *S*

The programming model of *S* is based on two main components: the deterministic control of any state change during the execution of the service and a simple yet powerful approach to parallelism.

Implicit State-oriented Programming. One of the major sources of complexity and performance degradation for parallel applications is the management of shared state. *S* solves this issue by forcing the developer to specify which operations will alter the state, and by decoupling the management of the state from the access to shared state. In *S*, different services do not share state by design. Resources within the same service may share state. The developer is thus forced to explicitly describe any possible interaction among services in terms of HTTP methods. In this way, the semantics of any HTTP method is enforced. Therefore, resources implementing the GET method will not be allowed to alter any

private or shared state, while resources implementing the POST method will be provided with complete access to both shared and private state. This clear division of state visibility, together with a clear separation of stateless and stateful operations, enables the compiler to explicitly control how the state of a RESTful Web service is accessed and manipulated.

Parallel Programming Model. *S* approaches parallel programming with a separation between what can be parallelized by the developer within the behavior of a specific request handler and what can be parallelized by the runtime to handle multiple concurrent requests.

The goal is to let developers write the service logic assuming that all the state will be and remain consistent no matter how many concurrent clients access the service. Furthermore, the developer does not have any control over the degree of parallelism used to execute the service as the runtime autonomously decides how many parallel processes should be allocated to run each service. In more detail, the runtime makes use of the implicit state-oriented programming assumptions to infer which parallelization strategy to apply. Therefore, read-only requests can be easily parallelized, while update-requests need to be serialized. Likewise, stateless services can be replicated, while stateful services can be replicated but their state needs to be kept synchronized by the runtime. The necessary locks and synchronization mechanisms are entirely managed by the runtime.

Conversely, developers can focus their parallelization efforts on reducing the overall response time of a request handler and on speeding up the interaction with external services. Such optimizations make use of two common high-level control-flow parallelism constructs which help to overlap the execution of multiple I/O operations. As we will show, these out-of-order parallelism constructs do not require the developer to reason in terms of threads or parallel processes, since the parallel execution of the instructions of the request handler is also managed by the runtime.

4. The *S* Service Scripting Language

4.1 Syntax

The *S* language is informally introduced with the examples in this section. The syntax of *S* is an extension of the JavaScript syntax; thus, any valid JavaScript statement can be used in *S*, with some limitations introduced to comply with the implicit state-oriented programming model. The JavaScript syntax has been extended to enable explicit parallelism statements such as out-of-order execution of I/O-bound operations, and explicit interaction with external HTTP resources.

The main entity of the language is the *service*. Services have local scoping, which means that two different services cannot share any global variable. Instead, since each service entity corresponds to an independently managed Web service at runtime, two service entities can communicate via HTTP.

Each `service` statement can declare one or more *resources* with the corresponding *request handlers*. Request handlers represent the event-driven entry point for programming the Web service behavior. Any request to the resources associated with a service is processed by a specific handler, defined using the `on` statement. Request handlers can react to any of the HTTP methods (such as GET, PUT, DELETE, and POST), and are associated with a unique resource identifier, specified with the `res` keyword. Services can have multiple request handlers. Following our state-oriented programming model, the scoping of handlers is also local: variables declared within the scope of a handler cannot be accessed from another request handler.

```
    // To be invoked with:
    //   GET /data
    //   PUT /data?value='...'

1   service helloWorld {
2      state shared = 'World'
3      res '/data' on GET { respond 'Hello ' + shared }
4      res '/data' on PUT { shared = query.value }
5   }
```

Listing 1. Simple stateful service in *S*.

Stateless (or purely functional) handlers can be associated with read-only GET methods. This implies that any JavaScript function invoked within GET request-handlers must be side-effects free. Since the other HTTP methods could alter the state of resources, the language supports the implementation of stateful request handlers in the following way. Request handlers sharing the same resource name (i.e., the same URL) with the need of a shared state can declare special static variables, identified by the `state` keyword. Such variables do not lose their state once the request handler has been invoked, and have their visibility limited within the scope of the declaring handlers.

State variables declared within an `on` construct are only addressable within that request handler, while state variables declared within the scope of a `service` block (or a `res` block with multiple `on` blocks) are accessible to request handlers sharing the same URL path. No state can be shared among different URL paths.

Handlers accessing the shared state do not have to implement any synchronization mechanism, and the consistency of the shared state is guaranteed by the runtime.

Finally, according to the HTTP specification, HTTP POST methods can cause the creation of new resources within the service. The language supports the creation of new resources at runtime by embedding nested resource declarations within request handlers. Such nested resources will be instantiated once the execution reaches their declaration point and will remain available to clients as long as they are not deleted.

4.2 A Simple Stateful Service

A simple Web service written in *S* is shown in Listing 1. The code corresponds to a simple "Hello World" service with a shared state (`shared`) and two request handlers: `on GET` and `on PUT`. When receiving HTTP GET requests (on the `/data` URL), the corresponding handler accesses the shared state and responds with the "Hello World" string. Since the GET method is idempotent and safe, the language allows GET handlers neither to modify any shared state, nor to declare any local state variable. This enables the runtime to execute multiple GET request handlers in parallel. The runtime system (and not the developer) is responsible for managing synchronization upon access to the shared state, and no explicit locking is required.

The service also implements the HTTP PUT request handler to modify the state of a resource. When receiving a PUT request (to the full path specified through the `res` keyword, e.g., `/data?value='universe'`), the value stored in the shared state is altered. Due to the stateful nature of this request handler, which implies that the state `shared` will change for every new PUT request, the runtime system

```
// To be invoked with:
//   GET /search?q=...

1  service proxy {
2     res '/search' on GET {
3        res g = 'http://google.ch/search?q=@'
4        if(query.q)
5           respond g.get(query.q)
6        else
7           respond 'Invalid query'
8     }
9  }
```

Listing 2. Simple proxy service.

cannot execute multiple PUT handlers in parallel, therefore, requests of this class are processed sequentially (fairness is not guaranteed). Also, concurrent GET requests cannot be processed consistently while a PUT handler is altering the shared state. The runtime system is therefore responsible for managing the parallel execution of GET handlers, that will answer with an outdated version of the shared state, and for updating the version of the state once its value will have been modified. The implicit locking runtime mechanism is aware of the actual state of the data elements that are shared. Thus, two different PUT handlers operating on two different states will be executed in parallel with no serialization.

In the previous example, the PUT handler uses a local object called `query`. This is a special object automatically created and managed by the runtime, containing all the information relative to the incoming HTTP request. The object is generated for each new request and its visibility is limited to one handler body scope. *S* provides another object with similar purpose, called `response`, managing every aspect of the response (e.g., HTTP headers, status codes, etc.).

4.3 RESTful Service Composition and Dynamic Nested Resources

Server-side applications often need to interact with external services, becoming a composition of existing services exposed as a new service. In *S*, the resources of external services are first-class entities, also defined through the `res` keyword.

The code of Listing 2 describes a proxy service receiving an input value (`q`) to be forwarded to another RESTful Web service (the Google search engine). In the code, `g` is an *external resource* managed by an external Web service. To support a complete binding between the URL addressing the external service and the corresponding entity in *S*, the resource can be declared using one or more `@` placeholders. In the example, this solution allows mapping the first argument of `g.get()` to the first parameter of the URL's query (i.e., `q`). When multiple parameters are expected, multiple `@` symbols can be used.

Having external resources as first-order entities makes the composition of external services straightforward. The example in Listing 3 presents a meta-search service which composes two popular search engines (Google and Microsoft Bing). As opposed to returning the results of the search as a response to the request (as done in the example in Listing 2), the code associates the result with a dynamically created resource (`/total/{id}`) and the client is redirected to it. To do so, the client invokes the server like in the previous examples, but instead of receiving a direct response, it receives an HTTP 302 code (redirect) pointing to the newly created resource containing the combined results from the two searches, which can be read using a GET request.

```
// To be invoked with:
//   POST /search?q=...
//   GET /total/{id}

1  service composition {
2     res '/search' on POST {
3        function combine(a,b) { ... }
4        state id = 0; id++
5        res g = 'http://google.ch/search?q=@'
6        res b = 'http://bing.com/search?q=@'
7        var total =
            combine(g.get(query.q),b.get(query.q))
8        res '/total/'+id {
9           on GET { respond total }
10       }
11       respond { 302 : { Location : '/total/'+id }}
12    }
13 }
```

Listing 3. Service composition in *S*.

This example also shows other interesting aspects. First, it uses a local state variable, called `id`. As described in the previous section, state variables can be used to manage a persistent state in request handlers that do not have to be idempotent. This is the case of the POST method, which is used to create new resources according to the HTTP specification. The semantics of state variables is straightforward: when declared, they can be initialized with a given value (`id = 0`, in our example). Successive invocations of the same request handler will ignore the initialization. In this way, the first time the `on POST` handler is invoked the state variable is initialized to zero and incremented. At the second invocation, the state variable has a value of 1 that is incremented as a result of the `id++` operation, ensuring that a unique identifier for the newly created resource (line 8) is assigned.

Another relevant feature shown in the example are nested resources. A new resource is created using the `res` keyword. The resource is created within the scope of the resource handler, but with an independent URL path. The new resource is declared specifying the path it will refer to (in our example, a string composed of `/total/` plus the unique identifier managed through the state variable) and one or more request handlers for the new resource. The scoping strategy adopted for nested resources declaration is the following: the new nested resource is created with a snapshot of all the global and local variables accessible within the event handler construct at the moment of its creation. According to our example, for instance, the new nested resource `/total` is allowed to access the `total` variable. Thus, when the new resource is accessed by a client, it responds with the value of `total` at the moment of its creation.

4.4 Explicit Control-Flow Parallelism Constructs

Service composition is a key feature of *S*. However, the invocation of multiple independent services is an I/O-bound operation that usually implies non-negligible latency. *S* allows speeding up service composition by performing I/O-bound operations in parallel. To this end, the language exploits an out-of-order parallelism model through the `par` and the `pfor` constructs.

Every set of instructions included in a `par` block is evaluated with respect to its data dependencies and control-flow. Any set of instructions with no such dependencies is executed in parallel, while the others wait for their data dependencies to be satisfied. This approach preserves the data-flow semantics of the original code, while introducing a partial

```
// To be invoked with:
//   GET /search?q=...

1  service helloPar {
2     res '/search' on GET {
3        function combine(a,b) { ... }
4        par {
5           res g = 'http://google.ch/search?q=@'
6           res b = 'http://bing.com/search?q=@'
7           respond combine(g.get(query.q),
                            b.get(query.q))
8        }
9     }
10 }
```

Listing 4. Usage of the `par` construct in *S*.

control-flow ordering which enables the compiler to schedule independent I/O-bound operations to be processed concurrently.

In addition to this construct, the `pfor` construct enables the parallel execution of code over multiple elements of one same data collection. In this case, the compiler checks that no data dependency is present among the different iterations and, if possible, executes the body of the loop in parallel for any element of the collection. Finally, `par` constructs can be nested into `pfor` constructs to further increase the code parallelization.

Listing 4 shows an example service using the `par` construct. This service is a different version of the example already discussed in Listing 3. The example shows how easy it is to parallelize the block of instructions performing the invocation of external services. The *S* compiler automatically identifies that the statements at lines 5 and 6 can be executed in parallel, and performs the two operations concurrently, waiting for both to complete before executing the next statement at line 7. Function calls are considered as field accesses, which means that the body of the `combine` function will not be parallelized. Also, functions are assumed to be side-effects free.

5. Compile-time Support

The result of the compilation process of an *S* program is a set of JavaScript source files. Each of these source files is passed to the *S* runtime which binds the compilation output to a set of parallel processes supporting the execution of implicit and explicit parallel operations.

5.1 Synchronous to Asynchronous Event-Driven Compilation

Any resource-related operation (i.e., any valid HTTP service invocation) is coded as a synchronous operation in *S*. This helps keeping the source code readable, and does not require the developer to write every service invocation as a set of complex nested callbacks, as it would be required when done in plain JavaScript. However, to fully exploit the benefits of the event-driven V8 runtime offered by Node.js, the JavaScript executable code generated by the compiler should be event-based, thus asynchronous. To address this issue, and to support synchronous service invocations in an asynchronous event-based runtime environment, the compiler uses the following de-synchronization strategy. While traversing the *S* Abstract Syntax Tree, the compiler performs these rewriting operations:

1) Each control-flow block performing I/O operations (e.g., constructs such as `if`, `for`, etc.) is compiled to a separate JavaScript function. JavaScript has static scoping: the scoping of any variable is preserved declaring all the global variables in an external block which will include all the inner levels.

2) Blocks containing an I/O operation are subdivided into two equivalent blocks: the first contains a runtime system call to trigger the beginning of the I/O operation. The latter contains an event handler that recovers the execution of the resource invocation when the I/O operation completes. The tree rewriting is done recursively (i.e., blocks with more than one I/O operation are subdivided into as many blocks as needed). Finally, each of the rewritten blocks is compiled into an independent JavaScript function.

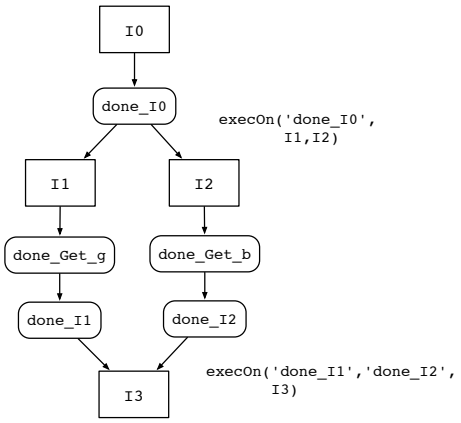
3) To preserve the original Control-Flow Graph (CFG) topology, all the compiled function blocks are enriched with an event notification mechanism. The runtime allows managing the asynchronous execution of any function associated with a specific event. The compiler appends an `emit()` runtime call at the end of each block. In this way, when a block completes its execution, the runtime is notified and can pass control to the next blocks.

4) The compiler computes a possible correct sequence of function calls triggered by event notifications (according to the CFG of the input code). These are mapped to invocations of another runtime component, called `scheduler`. Thanks to this component, the control-flow of the service can be driven according to specific events using a callback mechanism. By calling the runtime method `execOn(event, function)`, the runtime can tie the execution of a specific function as the consequence of a specific event. The compiler exploits this mechanism to reconstruct the correct control-flow of a request handler.

Overall, the CFG is converted to an Event-Driven Control Graph (EDCG), where every block is executed as a response to a triggered event and not as a consequence of a JavaScript control-flow construct evaluation. Whereas the final asynchronous code will be less efficient compared to its corresponding synchronous version, the advantage is that it becomes possible to use one process to overlap multiple parallel executions. This is important in our application domain, where a limited number of execution processes should scale to handle a very large number of concurrent clients.

5.2 Out-of-Order Parallel Execution

The compilation scheme adopted to transform synchronous code to its asynchronous executable version takes advantage of event-based function calling made available through Node.js. The event-based execution of different instruction blocks can be further exploited to implement the out-of-order parallel execution of I/O-bound operations. To this end, the compiler performs a static analysis of the source code identifying all the resource-related operations. For each `par`/`pfor` block, the compiler analyzes the contextual information relative to each variable access and modification, computing a per-instruction Data Dependency Graph (DDG). The graph is then compacted by clustering instructions which do not imply I/O-bound operations. At the end, the resulting DDG is traversed to reconstruct the correct calling tree and the corresponding scheduling instructions are emitted. The scheduler runtime object allows to schedule the execution of a specific block according to multiple events. At runtime, the scheduler suspends the execution until all parallel branches of the DDG are finished. In this way, the compiler guarantees that the data dependencies of the sequential execution are respected.




```
1   var G = {}
2   function combine(a,b) { ... }
3   var I0 = function() {
4      G.g = new runtime.resource('http://google.ch/search?q=@')
5      G.b = new runtime.resource('http://bing.com/search?q=@')
6      scheduler.emit('done_I0') }
7   var I1 = function() {
8        G.g.startGet(runtime.query.q)
9        G.g.on('done_Get_g', function()
           { scheduler.emit('done_I1') } }
10  var I2 = function() {
11       G.b.startGet(runtime.query.q)
12       G.b.on('done_Get_b', function()
           { scheduler.emit('done_I2') } }
13  var I3 = function() {
14       var t1 = G.g.resultGet()
15       var t2 = G.b.resultGet()
16       runtime.respond(combine(t1,t2)) }
17  scheduler.execOn('done_I0',I1,I2)
18  scheduler.execOn('done_I1','done_I2',I3)
19  scheduler.exec(I0)
```

Listing 5. Event-Driven Control Graph (left) corresponding to the JavaScript code (right) as produced by the *S* compiler for the source *S* code in Listing 4.

Listing 5 shows a portion of the JavaScript code as emitted by the *S* compiler. This code is the compiled version of the code snipped presented in Listing 4. The example shows how the synchronous-to-asynchronous compilation process and the `par` statement are converted into asynchronous event-based JavaScript. In the same figure, the EDCG of the *S* source is shown. Rounded boxes contain events, while square boxes describe a function call triggered by (incoming edge) or emitting (outgoing edge) a specific event. The interaction with the external resource (for instance `g.get()`) is compiled using the `runtime.resource` object, which emits an event (e.g., `done_Get_g`) when the remote invocation has completed. These events will trigger the `done_I1` and `done_I2` events asynchronously. Once both events have been triggered, the scheduling component of the runtime will resume the execution of the last block `I3` of the request handler, which can access the results fetched in parallel from the two resources.

6. The *S* Runtime

The JavaScript code generated by the compiler is executed by the *S* runtime. The runtime is running on top of the Google JavaScript V8 virtual machine and Node.js. We extended the V8+Node.js runtime with additional native modules, which provide support for load balancing, process control, and inter-process communication. Also, the runtime further extends the *S* language explicit parallelism mechanism (enabled through the `par/pfor` constructs) with implicit parallelism (handling multiple client requests in parallel). This implicit parallelism support is implemented in the runtime, based on a number of specialized concurrent processes, efficiently communicating through shared memory data channels or HTTP according to the context.

6.1 Resource Request Routing

By design, each *S* request handler is independent of the others in that its code can only access variables declared within its local scope or variables declared as shared state within the resource or the service. This allows the compiler to produce a separate JavaScript source file as output for each resource handler. These files are deployed for execution by the runtime, which publishes each service through its own TCP/IP address and port. To allow multiple request handlers to listen on the same port, the runtime makes use of another independent process, called *request router* (RR). The RR is responsible for opening the TCP connection port of the service, and for accepting HTTP requests from external clients. As soon as a new client request is received, the RR process accepts it and extracts the routing information by parsing the resource URL and HTTP method of the HTTP request header. With it, the RR identifies the request handler process responsible for the specific request URL.

To do so, the RR manages a routing table listing the mapping between resource URLs, methods, and process identifiers of the corresponding request handler processes. In order to deal with nested resources, the routing table is dynamically managed and updated as soon as new nested resources are added to a service.

The management of shared state relies on a similar approach. Each shared state variable is managed by an independent autonomous state manager process responsible for ensuring the consistency of the state as it is exposed to concurrent requests. All the stateful request handlers are processed by the state manager, while stateless requests are processed by external processes holding a cached version of the state. The semantics of the HTTP uniform interface enables each state manager to process multiple idempotent and safe requests in parallel.

Similarly, pure stateful request handlers (i.e., request handlers accepting only PUT, POST, and DELETE methods) also use a state manager. However, since no stateless operations are present, the state is not replicated among different processes, but it is located within the process hosting the request handlers.

Nested resources are also managed as independent processes, like any regular request handler. The only difference is that nested resources are dynamically registered and unregistered from the service routing table.

Since external RESTful services and different request handlers correspond to the same entity in *S* (managed through the `res` keyword), the runtime system is responsible for dynamically resolving each resource's address and using the appropriate communication mechanism. Therefore, communications between different resources within the same ser-

```
// To be invoked with:
//   PUT /crawl?startFrom=...&depth=...
//   GET /urlsDiscovered

service crawler {
   res '/crawl' on PUT {
      function scan(page) { ... }
      res url = query.startFrom
      var list = scan(url.get())
      if(query.depth>1)
         pfor(var i in list) {
            par {
               res crawl =
                     '/crawl?startFrom=@&depth=@'
               res discovered =
                     '/urlsDiscovered?val=@'
               crawl.put(list[i],query.depth-1)
               discovered.put(list[i])
            } }
   }
   res '/urlsDiscovered' {
      state urls = new Array()
      on PUT {
         urls.push(query.val)
      }
      on GET {
         respond urls
      }
} }
```

Listing 6. Parallel Web Crawler in *S*.

```
// To be invoked with:
//   POST /start?urls=...&key=...
//   GET /red/...

service mapred {
   res '/start' on POST {
      state id = 0; id++
      res '/red/'+id {
         state s = 0
         on GET { respond s }
         on PUT { s += query.count }
      }
      respond { 302 :
         { Location : '/red/'+id } }
      pfor(var i in query.urls) {
            res scan = '/map?url=@&key=@&id=@'
            scan.put(query.urls[i], query.key, id)
      }
   }
   res '/map' on PUT {
      function scan(page,key) { ... }
      res url = query.url
      var page = url.get()
      res reduce = '/red/'+query.id+'?count=@'
      reduce.put(scan(page, query.key))
   }
}
```

Listing 7. Map-Reduce in *S*.

vice are carried out through shared memory communication channels, while communications with external services are carried out with standard TCP sockets and HTTP.

6.2 Parallel Runtime Architecture

As discussed in Section 2.1, the semantics of the HTTP protocol allows processing multiple requests to the same resource in parallel. In more detail, *S* concurrently processes multiple requests of some type (like GET), while others (like POST) require exclusive access. Following the design of the *S* language, also requests associated with stateless handlers can be easily parallelized. Only request handlers altering a private state cannot be executed in parallel to ensure consistency.

To identify which request handlers can be executed in parallel, in conjunction to the HTTP semantics, *S* leverages a compile-time static analysis. The analysis is based on the verification of access patterns to shared states. When (at compile-time) a stateless resource is found, it is marked with a special identifier allowing the runtime to parallelize the execution of the specific handler

Request handlers that cannot be parallelized are executed as a single process by the runtime system. Depending on the available hardware resources, other handlers are replicated among multiple processes and requests are automatically load-balanced among them by the so-called *Stateless Resource Manager* (SLR).

7. Case Studies

Complex services requiring the interaction with several external services can be easily implemented in *S*. In this section, we illustrate two common Web services developed in *S*. The scalability of the two services will be evaluated in Section 8.

The first case study demonstrates a self-parallelizing Web crawler. The service implementation shows the flexibility of the language regarding service composition. The crawler service composes services by crawling external Web pages, and by recursively calling itself. The example also shows how stateless request handlers can benefit from parallel execution without having the developer to deal with process-level parallelism.

The second example is a Map-Reduce service operating on external resources. The service features several parallel components, resulting in a complex runtime architecture.

7.1 Web Crawler Service

The source code of the Web crawler case study is presented in Listing 6. The service recursively traverses a set of linked HTML pages and collects their URLs. It is composed of two resources, one used to crawl a Web page and the other (`/urlsDiscovered`) used to collect and to publish the results of a crawl.

The service is invoked with a PUT request on the `/crawl` resource. The corresponding request handler downloads the first external resource (a Web page specified by the client in the request with the `startFrom` parameter), and calls the `scan` method. This function implements a simple HTML parser which scans the given input data (containing the downloaded Web page) and returns an array containing all the URLs found. Then, for each URL contained in the `list`

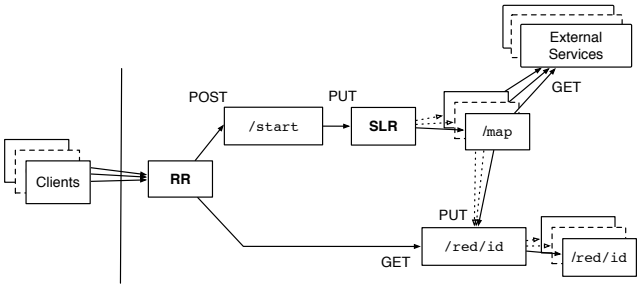


Figure 1. Runtime architecture of the Map-Reduce Web service case study as executed by the *S* runtime system.

array, the service recursively calls itself to scan for further URLs. Meanwhile, all the URLs identified by the service are saved. The `/crawl` request handler is kept stateless by sending a PUT request to the `/urlsDiscovered` resource.

Since the `/crawl` resource is stateless (that is, it is not directly changing a shared state nor maintaining a local state), the runtime can handle multiple requests in parallel. The stateless nature of the handler, coupled with the usage of the `pfor` construct, automatically parallelizes the execution of the resource handler. In fact, for each set of URLs found at any recursive invocation, the `crawl` handler receives multiple parallel requests generated by itself. Since the runtime does not make any difference between requests coming from clients or from internal request handlers, the `crawl` handler reacts as if it would have to respond to an increasing number of client requests, and will thus use an increasingly larger amount of execution resources, effectively parallelizing the Web crawling operation.

7.2 Map-Reduce Service

Map-Reduce computations are composed of a parallel computation (the "map" function) followed by a sequential gathering operation (the "reduce" function). Listing 7 shows how this can be implemented in *S*. The Map-Reduce service publishes a `/start` resource, which receives as input a keyword and a list of external URLs corresponding to a list of Web pages (for instance, the list of Web pages could be the result of the computation performed by the crawler Web service presented before). Right after a client POST request is handled, like for many common Map-Reduce applications. The service applies a function (`scan`) which is executed in parallel and counts the number of occurrences of the given keyword in all the Web pages. Finally, the result is stored in a shared state, and is made available through a new nested resource, so that each client will have a personalized result.

The example exploits the two parallelism models provided by *S* and presents a complex runtime architecture, including nested resources and shared state management.

First, at line 10, the `pfor` construct is used to parallelize the download of all the URLs received as input. In this way, the `/start` resource can invoke the `/map` resource multiple times in parallel. This is possible thanks to the static analysis performed by the *S* compiler. Since the `/map` resource is stateless (it only has a PUT request handler with no private state) the runtime can execute multiple requests to `/map` concurrently.

An overview of the runtime architecture used to run the Map-Reduce service example is shown in Figure 1. Each box in the figure represents an independent process. Arrows indicate routing paths as specified by the *S* runtime, while dotted boxes represent processes that can be dynamically parallelized by the runtime, determining the right number of parallel processes to be executed on the fly based on the available resources.

8. Performance Evaluation

S has been designed to enable the development of high performance RESTful services. In this section, we provide an evaluation of the performance of the two case studies presented in the prior section. Our results clearly demonstrate that services written in *S* can benefit from the parallel runtime architecture and scale to handle thousands of concurrent client requests when deployed on multicore machines.

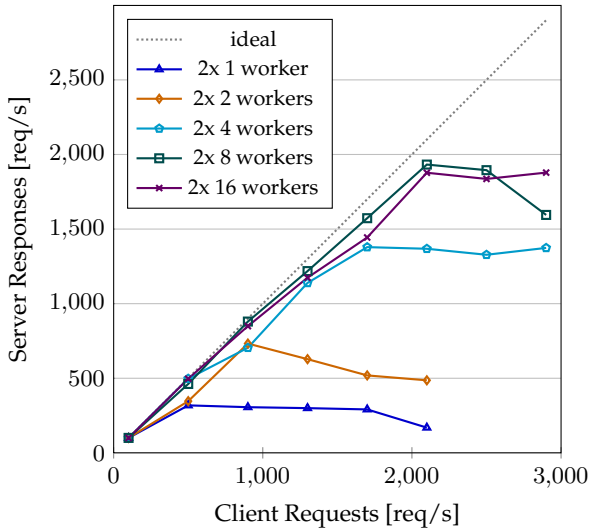


Figure 2. Map-Reduce scalability experiment.

8.1 Map-Reduce Service

The scalability of the *S* Map-Reduce service has been evaluated through the following experiment measuring how well it can use an increasing number of CPU cores to serve an increasing number of clients. The service has been deployed on a server machine with a total of 24 cores.

The experiment has been executed with the `/map` resource downloading a pool of Web pages hosted on a different machine. For each client request, the Map-Reduce service is requested to download a total of five Web pages in parallel and to count, for each page, the number of times a given keyword appears. The number of external pages to download has been set to such a low value to measure the service's scalability without any risk of network bandwidth saturation. Every incoming client request to the `/start` resource triggers five concurrent outgoing requests to the `/map` resource, which again performs an HTTP GET request to retrieve the given Web page.

Results are described in Figure 2. The experiment has been executed by configuring the service to use an increasing number of parallel workers, up to a maximum limit of 32 processes (16 parallel processes for the Map phase and 16 for the Reduce phase). The chart shows that this service scales almost linearly up to the limit of the physical resources available in the system.

8.2 Web Crawler Service

The algorithm implemented by the Web Crawler service forces the service to call itself recursively for each new URL found in the page currently being analyzed. This generates an increasing number of Web pages to be crawled for each iteration, corresponding to an increasing number of client requests for the service to be processed.

The service is started by a single request sent from the client, containing an URL from where to start the crawling process, and the level of recursion depth to halt the service at. Due to the nondeterministic nature of the Web, the performance of a Web crawler cannot be measured using real Web pages. For this reason, we have created a set of ad hoc Web pages representing a (potentially) infinite binary tree. The regular structure of the tree lets us use the number of

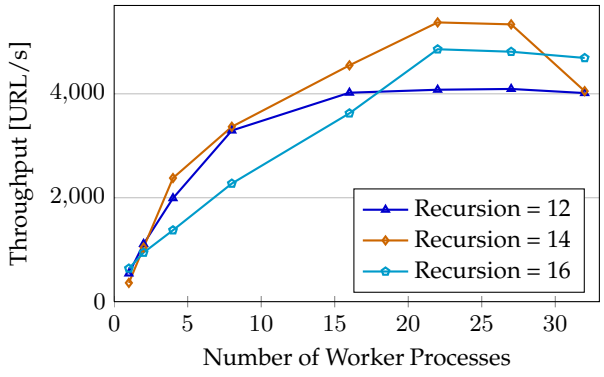


Figure 3. Crawler scalability with different recursion depths.

nodes crawled as a measure of the performance of the service. The crawler service is deployed on the same 24 cores machine of the previous experiment.

In Figure 3 the evaluation of the crawler service implemented in *S* is presented. The chart shows the average throughput of the service (number of crawled URLs per second) as obtained using different parallelism degrees. The three curves correspond to three different depth levels of the tree, namely 12, 14, and 16, corresponding to about 8.2×10^3, 3.3×10^4, and 1.3×10^5 Web pages to crawl.

Each curve has been obtained by increasing the maximum number of parallel resources allocated for the service. The chart in the figure clearly shows how the service is able to exploit the underlying parallel hardware up to the scalability limit imposed by the hardware resources available in the system.

Another evaluation of the parallel algorithm executed by the crawler service can be done considering the way the service visits the crawling tree over time. To this end, in Figure 4 the temporal evolution of the service is shown with regard to the number of requests received by the service, and the number of responses it has been able to return. The four charts present the evolution of these two metrics over time for four different upper bounds of parallel workers.

The chart with a single worker shows how the recursive tree traversal code makes the service continuously receive requests for new nodes to be crawled. Since only one worker process is processing the requests, the service can then begin to answer only when it has reached the crawling frontier (that is, it has visited all the leaf nodes of the tree). This is equivalent to the standard evolution of a sequential invocation of recursive functions: the call stack continues to grow until the recursion can be stopped.

More interesting things happen when the service is allowed to exploit parallel resources. As visualized in the remaining plots, the service is able to overlap the tree traversal with the answer reconstruction. This is mainly due to the associative nature of the crawling operation, as the result can be reconstructed independently of the order in which the tree is visited. Increasing the number of parallel workers available in the system both increases throughput and reduces the total execution time.

9. Related Work

The idea of exploiting state-related information to identify whether an application can be parallelized has been discussed in the context of the so-called Permission-Based pro-

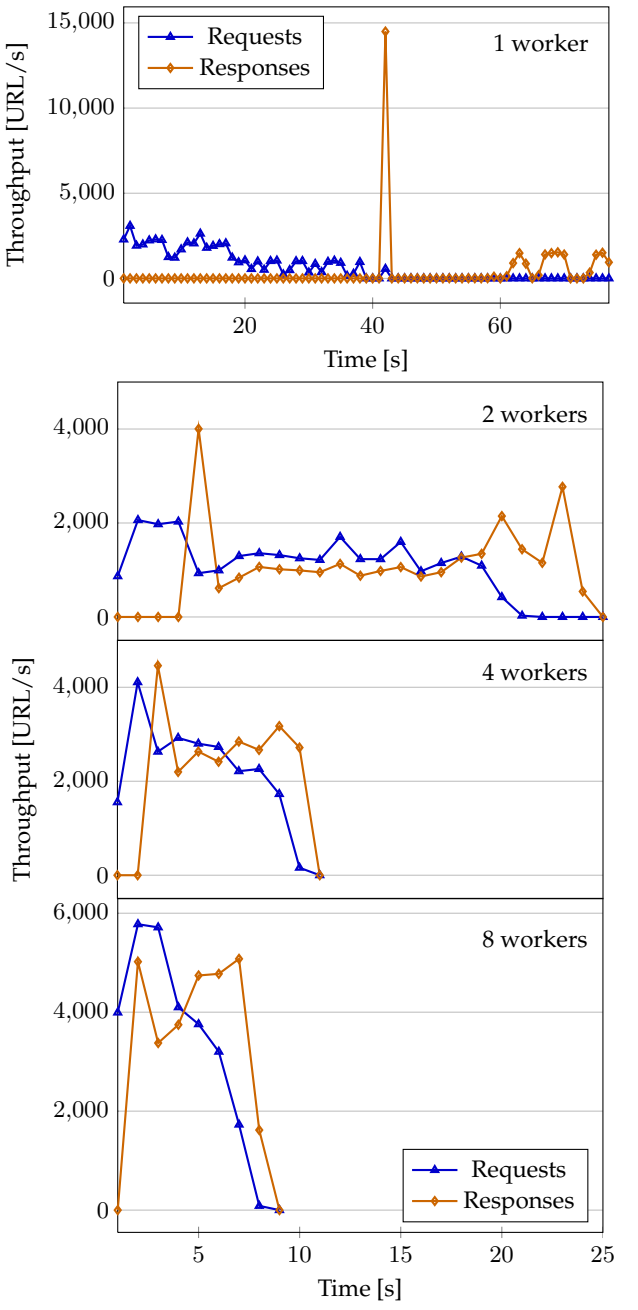


Figure 4. Crawler running with 1, 2, 4, and 8 workers.

gramming [4]. Permissions are annotations on variables and objects such as "read-only" [14] which can be used to address several issues in software engineering, including concurrency [6]. A relevant approach in the direction of Permission-Based programming languages is represented by the Plaid language [3]. Plaid is a programming language where concurrency is the rule, and dependencies between operations are specified using permissions, allowing the runtime to automatically execute applications concurrently. Similarly to Plaid, *S* exploits the way state is managed to infer which operations can be executed in parallel. As opposed to Plaid's approach, *S* state-related primitives are implicitly defined using the unique semantics of the HTTP protocol.

S resources and the corresponding request handlers are influenced by Actors [2]. Unlike traditional actor-based languages (e.g., Erlang [20] or Scala [12]) the Actor-based semantics in *S* is kept implicit, and the way a handler processes messages coming from other peers (or external clients) depends on the kind of method associated with the request.

Self-parallelizing runtime system is featured in several frameworks. An approach similar to the self-parallelizing strategy implemented in *S* is represented by the Self Replicating Object (SRO) programming model presented in [17]. SRO are objects able to partition their state to permit a parallel execution. Similarly to the implicit parallelism in *S*, SRO objects can identify whether the state of a component allows parallel execution or not. One relevant difference is that the *S* runtime does not partition state.

With aims similar to the ones behind the parallelism constructs of *S*, the programming model of OoO-Java enables the automatic parallelization of Java code through the out-of-order execution of data-independent instructions [15]. Differently from the general purpose of OoO-Java, *S* supports out-of-order execution of different operations only for I/O-bound operations. Other speculative models can be seen as an alternative to *S* parallelism constructs [5, 21], as well as notation-based models [7, 18]. Due to the peculiarity of the Web services domain, the solution of parallelizing I/O-bound operations adopted in *S* aims at combining the strengths of both approaches as it enables the developer to explicitly identify which part of the code should be parallelized (as with annotations) without any risk of altering the original sequential semantics (as guaranteed by deterministic speculative approaches).

Out of the realm of server-side development technologies, JavaScript-based approaches have been proposed in several cases. Flapjax [16], for instance, is a JavaScript-compatible language based on a Functional Reactive Programming model [8]. Similarly to *S*, Flapjax adopts an event-driven approach, and similarly to the *S* compiler, Flapjax code is compiled to JavaScript. Finally, an exhaustive performance analysis of several JavaScript applications is presented in [11].

10. Conclusion and Future Work

In this paper we presented the *S* service scripting language, its compiler, and its runtime system. The initial domain targeted by the language consists of service-oriented applications, for which a parallel runtime architecture suitable for high-performance execution of RESTful Web services has been developed. The language features explicit control-flow parallelism constructs which developers can apply to speed up the execution of individual request handlers. Additionally, the language leverages its implicit state-oriented programming model to automatically parallelize the execution of stateless and stateful services by employing desynchronization and self-parallelization techniques. The results presented in the paper show how applications developed using the *S* language can efficiently exploit parallel architectures such as multicore machines to scale in the number of clients they can serve concurrently.

A further extension of the *S* language and runtime system will deal with another class of modern Web services, namely HTTP-based streaming services. In such a context, the *S* self-parallelizing runtime will also have to deal with non regular streams, and will have to adapt to the actual frequency of incoming requests. We also have started to experiment with self-tuning mechanisms based on auto-scaling techniques which not only take into account the number of cores available on the machine but also dynamically adjust the allocation of processes to cores based on their actual workload.

Acknowledgments

The work presented in this paper has been supported by the Swiss National Science Foundation with the SOSOA project (SINERGIA grant nr. CRSI22_127386).

References

[1] HTTP protocol specification. URL `http://www.w3.org/Protocols/rfc2616/rfc2616.html`.

[2] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986.

[3] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proc. of OOPSLA*, pages 1015–1022, 2009.

[4] J. Aldrich, R. Garcia, M. Hahnenberg, M. Mohr, K. Naden, D. Saini, S. Stork, J. Sunshine, E. Tanter, and R. Wolff. Permission-based programming languages: Nier track. In *Proc. of ICSE*, pages 828–831, 2011.

[5] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Proc. of OOPSLA*, pages 81–96, 2009.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *Proc. of OOPSLA*, pages 211–230, 2002.

[7] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5: 46–55, January 1998.

[8] C. Elliott and P. Hudak. Functional reactive animation. In *Proc. of ICFP*, pages 263–273, 1997.

[9] A. Fettig and G. Lefkowitz. *Twisted network programming essentials*. O'Reilly, 2005.

[10] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[11] E. Fortuna, O. Anderson, L. Ceze, and S. Eggers. A limit study of JavaScript parallelism. In *Proc. of IISWC*, pages 1–10, 2010.

[12] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410:202–220, February 2009.

[13] I. Hickson. Web workers. World Wide Web Consortium, Working Draft WD-workers-20110310, March 2011.

[14] J. Hogg. Islands: aliasing protection in object-oriented languages. In *Proc. of OOPSLA*, pages 271–285, 1991.

[15] J. C. Jenista, Y. h. Eom, and B. C. Demsky. OoOJava: software out-of-order execution. In *Proc. of PPoPP*, pages 57–68, 2011.

[16] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proc. of OOPSLA*, pages 1–20, 2009.

[17] K. Ostrowski, C. Sakoda, and K. Birman. Self-replicating objects for multicore platforms. In *Proc. of ECOOP*, pages 452–477, 2010.

[18] K. H. Randall. *Cilk: efficient multithreaded computing*. PhD thesis, 1998.

[19] S. Tilkov and S. Vinoski. Node.js: Using JavaScript to build high-performance network programs. *IEEE Internet Computing*, 14:80–83, November 2010.

[20] R. Virding, C. Wikström, and M. Williams. *Concurrent programming in ERLANG (2nd ed.)*. Prentice Hall International (UK) Ltd., 1996.

[21] C. von Praun, L. Ceze, and C. Caşcaval. Implicit parallelism with ordered transactions. In *Proc. of PPoPP*, pages 79–89, 2007.