GEMs: Shared-Memory Parallel Programming for Node.js

Daniele Bonetta
Oracle Labs,
Austria
daniele.bonetta@oracle.com

Luca Salucci
Università della Svizzera italiana,
Switzerland
luca.salucci@usi.ch

Stefan Marr
Johannes Kepler University Linz,
Austria
stefan.marr@jku.at

Walter Binder
Università della Svizzera italiana,
Switzerland
walter.binder@usi.ch

Abstract

JavaScript is the most popular programming language for client-side Web applications, and Node.js has popularized the language for server-side computing, too. In this domain, the minimal support for parallel programming remains however a major limitation. In this paper we introduce a novel parallel programming abstraction called *Generic Messages* (GEMs). GEMs allow one to combine message passing and shared-memory parallelism, extending the classes of parallel applications that can be built with Node.js. GEMs have customizable semantics and enable several forms of thread safety, isolation, and concurrency control. GEMs are designed as convenient JavaScript abstractions that expose high-level and safe parallelism models to the developer. Experiments show that GEMs outperform equivalent Node.js applications thanks to their usage of shared memory.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming

Keywords JavaScript, Node.js, Generic Messages.

1. Introduction

In contrast to other languages, JavaScript was not designed to express parallelism. This is a clear limitation for cloud computing and data-intensive applications; domains in which the language has been popularized by Node.js [46].

Bringing parallel execution to a non-parallel language is challenging. Beyond few notable research efforts (e. g., the RiverTrail [28] data-parallel model), WebWorkers [4] is the only parallelism model for client-side and server-side JavaScript. The WebWorkers model is inspired by Actors [11], and is based on share-nothing fully-isolated parallel entities (i.e., workers) exchanging data via asynchronous messages. Although such form of share-nothing parallelism is a good fit for, e. g., stateless Web services, it prevents developers from taking full advantage of the shared memory available in modern server-class multicore machines. Moreover, share-nothing parallelism forces developers to explicitly partition and distribute data [36], and requires the usage of external services such as Memcached [1] when shared state is needed.

For data-intensive applications and so-called microservices [48], shared memory can be employed efficiently in several ways. For example, it can be used to implement in-memory caching for scale-up services [14], to optimize in-memory parallel processing for data-intensive applications [50], as well as to optimize the communication mechanisms used by microservices frameworks such as Amazon Lambda [5] via in-memory data transfer and zero-copy messaging.

Shared-memory parallel programming, however, is hard, as it requires developers to deal with data races and synchronization [36]. For Node.js, explicit shared-memory programming models such as the one of Java might be even more problematic, as programmers are used to—and existing libraries are built for—the event-based race-free model enforced by WebWorkers.

In this paper, we introduce a new parallel programming abstraction called *Generic Messages* (GEMs), specifically designed to enable selected forms of shared-memory parallelism in the context of the WebWorkers model. Informally, GEMs are a new form of messages that can be exchanged between workers to provide *controlled* access to shared-memory programming abstractions. Rather than enabling a

specific parallel programming model, generic messages are customizable and can be used to expose shared memory to workers in several ways. In particular, GEMs enable *safe* parallel programming models for WebWorkers, without exposing developers to low-level issues such as data races.

This paper makes the following contributions:

(1) It describes the Generic Messages model, a new parallel programming abstraction to enable shared-memory parallelism for share-nothing, fully-isolated models such as WebWorkers.

(2) It describes how Generic Messages can be used to implement six types of GEMs corresponding to six well-known parallel programming models. Each GEM enables shared-memory programming in Node.js by allowing developers to extend existing message-based applications with programming abstractions that can enforce thread-safe access to shared memory.

(3) It describes the implementation of the GEMs model in the context of Node.js, and we evaluate several Node.js applications using GEMs, highlighting the performance benefits, along with other benefits such as thread safety, over plain Node.js applications based on WebWorkers.

This paper is structured as follows. In Section 2 and Section 3 we motivate the GEMs programming model, which we present in detail in Section 4 and Section 5. In Section 6 and Section 7 we evaluate the performance of GEMs. Section 8 discusses related work, and Section 9 concludes.

2. Isolated Communicating Workers

The only model for parallel execution supported by Node.js is based on isolated parallel entities, called *workers*, which communicate using asynchronous message passing. One of the reasons for this is JavaScript's single-threaded language design. Since JavaScript is single-threaded, the Node.js runtime (and the underlying Google V8 virtual machine [3]) are single-threaded as well. Consequently, workers do not support shared memory, neither in the programming model (since each worker has a fully isolated memory space), nor at the level of the runtime system (because internal data structures are not thread-safe and the garbage collector requires the heaps of workers to be disjunct). In Node.js, workers can be used via a module called *Cluster* [8], which provides the basic support for messaging, as well as a Node.js-specific mechanism to let multiple workers listen on the same HTTP/TCP port. The Cluster module is designed to *scale-up* Node.js services within a single multicore machine, and performs automatic load balancing between workers listening on the same TCP port.

Share-nothing parallelism is an ideal model for several applications, e. g., scatter/gather parallelism [17]. For cloud and data-intensive workloads, however, the absence of shared memory can be a limitation. For some problems, shared memory is a more natural solution [45], and forcing developers to model every interaction with asynchronous message passing increases complexity when atomicity and consistency are required. For instance, since Node.js workers cannot share any resource at runtime, they need to use external systems such as Memcached [1] whenever shared state is needed. Although such caching systems offer properties such as distribution (over a cluster) and failure tolerance, they also result in additional data lookup overheads. As a consequence, Node.js applications often make use of a per-process, replicated, temporary cache to store data in the Node.js memory space (e.g., using external modules such as TTL [9]) in order to reduce the overall Web service latency. Especially for data-intensive applications, the overhead of moving data between Node.js processes and an external caching system can be prohibitive. Instead, a simple and efficient shared-memory solution is desirable to enable scalability within a single multicore machine, as it would remove the need to replicate objects in the memory space of the external caching service. Moreover, it would not require Node.js developers to program against a foreign API. From the perspective of the runtime system, shared memory can have benefits, too. For example, web services and big data applications can take advantage of shared in-memory data structures to reduce latency and improve data locality, avoiding the overhead and the complexity of replication systems that need to guarantee data consistency for non-shared-memory systems. Specifically for Node.js, in-memory communication can also be used to optimize message passing between workers, to avoid the current send-by-copy approach, which requires JSON data marshalling for every message.

GEMs enable the use of shared memory both implicitly and explicitly. They can be used to optimize message-based communications by implicitly using shared-memory mechanisms [25, 41] (without modifying the user-level worker API), and can also be used to expose safe, high-level, parallel programming models that combine message passing and shared memory.

3. GEMs to the Rescue

Informally, a GEM is a runtime mechanism to mediate access to a shared object graph between two or more parallel workers. GEMs can be considered a form of software capability [40] which defines how data are shared, and how they can be accessed. The way such access is controlled is defined by a GEM itself, and cannot be altered by the workers. Thus, a GEM can give workers *controlled* and *thread-safe* access to a shared object graph. GEMs are exposed to workers in the form of standard JavaScript objects, for which they can read and write properties, execute function calls, and perform other common operations.

A Motivating Example. As a concrete example for GEMs, let us consider a data scraping application that processes text to compute the frequency of keywords in a file. A parallel Node.js implementation for this example is depicted in

Master

```
var registry = new BigObject();
var keys = ["some","Important","words"];
// send data to workers
workers.multicast(keys)
  .scatter(registry)
  // callback when all workers replied
  .gather(function(results) {
    var total = accumulate(results);
    console.log('result is' + total);
  });
```

Workers

```
// callbacks for multicast and scatter
worker.on('multicast', function(msg) {
  keys = msg;
})
.on('scatter', function(msg) {
  var registry = msg;
  var result = {};
  // scan received part of document
  for (var k in keys) {
    var total = 0;
    for (var token in registry)
      if (registry[token] == keys[k])
        total++;
      result[keys[k]] = total;
  }
  worker.reply(result);
});
```

Figure 1. Node.js-compatible implementation of a *Wordcount* benchmark.

Figure 1. In this application, a first worker called "master" initially owns the data for the `document` to be scanned (previously read from a file) and the `keys` to be searched for. To enable parallel processing, the master partitions the input data, and sends each partition to parallel workers. The document is partitioned by the master using the `scatter` function, while the keys are copied to each worker using the `multicast` function. Finally, the `gather` function is called once all the results from each worker have been collected.

The example highlights some of the drawbacks of the worker model in Node.js: each time the master has to exchange data with the workers, it has to deep copy the data from its memory space to the memory space of the receiver. Here, this means that the `keys` array and the `registry` object are replicated for each worker. Copying and transferring the data reduces the performance of the application and increases its memory footprint. Intuitively, a shared-memory-based implementation would minimize communication by sharing only a reference to `registry` and to the `keys` array between workers. Sharing a direct pointer to such data, however, would expose workers to potential race conditions, as workers will be granted the *right* to perform concurrent writes.

Master

```
// create three Gems
var atomic = require('atomic-gem'),
  part = require('partitioned-gem'),
  readonly = require('read-only-gem'),
  registry = readonly.create(
                      new BigObject());
var keys = part.create(
          ["some","Important","words"]);
var finalResult = atomic.create({});
// multicast three Gems to the workers
workers.multicast(
    registry, keys, finalResult)
  .gather(function() {
    // no need for post-processing
    console.log(finalResult);
  });
```

Workers

```
// callback executed when the
// three Gems are received
worker.on('multicast', function(
    registry, keys, finalResult) {
  for (var k in keys.subset())
    for (var token in registry)
      if (registry[token] == keys[k])
        finalResult.atomic(function() {
          finalResult[k] =
            finalResult[k]+1 || 1;
        });
  worker.reply('done!');
});
```

Figure 2. An implementation of a *Wordcount* benchmark that benefit from using multiple GEMs.

Such race conditions can be avoided using GEMs for each of the two objects to be shared (i. e., `registry` and `keys`). Specifically, the two objects can be shared between workers using a GEM granting (and enforcing) read-only access. In this way, workers can receive direct access to the two shared objects, minimizing the data-transfer overhead. The code in Figure 1 can already use such kinds of GEMs without any changes, since no write operations are performed on the received messages. To this end, the `scatter` function would need to *create* a GEM for the `registry` and `keys` objects, and use it for the communication rather than copying and partitioning the two objects.

***Advanced Semantics for* GEMs.** GEMs can have more advanced semantics than read-only protection. They can expose arbitrary APIs for controlling access to the shared object graph, and can provide selective access rights to shared objects in multiple forms, for instance by granting read and write access only to a subset of the elements or properties in a shared object graph. Another version of the example using other, more advanced, types of GEMs is depicted in Figure 2. In this second implementation, the `keys` array is sent to all workers using a GEM that enforces *partitioned* access control. This GEM enriches the object graph that is being

shared with the `subset` function as an additional API, which can be used by the worker to claim read and write access limited to a partition of the shared `keys` array. By invoking `subset`, the GEM automatically assigns (in a thread-safe way) a subset of the elements of `keys` for exclusive access to a single worker, ensuring that other workers trying to read from (or write to) one of the elements of the array will not be allowed to do so. The GEM in the example not only provides partitioned access to `keys`, it also *enforces* it: any attempt to access any non-granted element of the array, i. e., not obtained via `subset`, results in an exception. The example also makes use of a second kind of GEM, the `results` object. This GEM enriches the object capabilities with the `atomic` API, which can be used to perform a sequence of operations on the shared object graph in a thread-safe way. The object is initially empty, and is used in this example to accumulate the final result of the scraping, thus avoiding the final accumulation operation on the master, as it was done in the Node.js implementation in Figure 1. The GEMs described in this example ensure thread safety for the GEM user, i. e., the worker. The way thread safety is achieved and concretely implemented is GEM-specific, as every GEM can provide different safety guarantees.

4. GEMs Design and Implementation in Node.js

A GEM is a new type of JavaScript object that can be exchanged between workers via message-based interactions to enable shared-memory parallel programming. The GEM model does not specify *how* shared memory should be exposed to workers. For example, it can be introduced implicitly, as in the example of Figure 1, where concurrency control mechanisms are transparently encapsulated in the GEM at the granularity of property reads and writes. Shared-memory programming can also be exposed explicitly, by providing custom APIs such as the `atomic` function described in Figure 2. Such a model-agnostic design makes it possible to trade off convenience, performance, and safety at a high granularity.

At the high level, a GEM is a combination of the following two elements:

(1) A *shared object graph*, that is, an object graph to be shared with multiple workers. The elements composing the graph can have any valid JavaScript value (e.g., Numbers and Strings). However, they can be associated only with one GEM, i.e., it is not possible for two GEMs to reference the same object graph, neither directly nor indirectly.

(2) *Dynamic sharing semantics*, which controls *how* the shared object graph can be accessed in parallel by multiple workers.

With Node.js being a share-nothing framework, GEMs can initially be exchanged only by an explicit message that is sent from a Node.js worker (called the GEM *owner*) to all the workers that need access to it in the parallel application. Using message passing as the core mechanism for sharing GEMs across workers allows developers to add shared-memory parallel programming selectively, without breaking existing applications. GEMs are not available by default, but they can be imported in the form of Node.js modules. In the following sections we describe the API used to define such modules together with the concrete internal structure of GEMs.

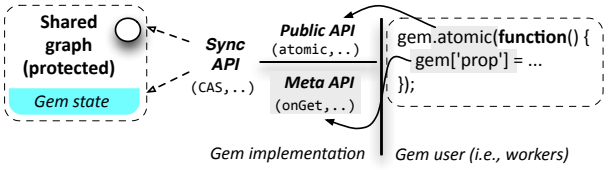

Figure 3. Overview of GEMs' internal structure.

4.1 GEM Definition and Creation

Generic messages are designed to be reusable. To this end, the model provides a specific API that is only available for the implementation of a GEM, but is not accessible by GEM users. Generally, for the implementation of a new type of GEM, a deep understanding of concurrency is required. Thus, we clearly distinguish the expert role of implementing GEMs, from the role of a GEM user who builds high-level applications based on readily available libraries of GEMs but does not need to be a parallel-programming expert. Therefore, we clearly distinguish between defining a new *type* of GEM, and the creation of a GEM instance (of a given type). To enforce the desired separation of roles, these built-ins are only available for Node.js module developers, which is detailed in Section 4.3.

New types of GEMs can be defined using the `Gem.define` built-in function. `Gem.define` is used as in the following example:

```
// Define a new type of Generic Message
var readOnly = Gem.define(gemConfiguration);
// Export it to Node.js
module.exports.gem = readOnly;
```

New types of GEMs can be created *only* from within Node.js modules definitions. This implies that new GEM types cannot be created explicitly by Node.js applications, but must be created and packaged in the form of a module. This also implies that the semantics of a GEM cannot be altered by its users, and ensures that the thread-safety guarantees provided by a GEM cannot be altered during its usage. The `define` function expects an object to be provided as argument. This object is called the GEM *configuration*, and is

a special JavaScript object that is used to specify the runtime semantics of the GEM for parallel and concurrent access.

To import a GEM of a given type in a Node.js application, an instance of the GEM has to be created. This can be done by importing a GEM using the standard Node.js package management system. After the GEM module has been loaded, a new instance of the GEM can be obtained by calling `Gem.create`, as described in the following example:

```
// import the gem from a module
var readOnly = require('read-only-gem');
// object to be used as the 'shared graph'
var content = {some:0,values:1};
// create an instance of the gem
var gem = readOnly.create(content);
// send the gem message to workers
for (var w in workers)
    workers[w].send(gem);
// object and gem are distinct
content.some++;
gem.some == content.some; // false
// this gem enforces strict
// read-only access on its state
gem.some++; // throws exception!
```

After the GEM has been created, it can be shared with workers via message passing. The GEM constructor accepts an optional input parameter (the `content` object in the example). This object can be used as the initial state of the GEM, and corresponds to the state that will be shared with all the workers that receive the GEM. The content object is optional, and other GEMs can have a GEM-specific way to allocate and modify the state to be shared. When such an initial shared object is provided, its entire object graph is copied, and the copied graph is made private to the GEM. The copying mechanism is called *JSON copy*, as it relies on the semantics of JSON object serialization [7]. Specifically, a JSON copy of a JavaScript object graph corresponds to a new object graph that has been built by (1) encoding the original graph in JSON format, and by (2) de-serializing the encoded graph to a new JavaScript object. As a consequence of JSON copy, the GEM-private shared object graph owns only the raw data and the structure of the original object graph. It neither includes functions nor JavaScript-specific features such as the object prototype chain of the original object graph. Moreover, cycles in the shared object graph are removed as they are not supported by JSON.

Using JSON data as the format for the initial state to be shared between workers has two main motivations. First, it ensures that the same shared object graph cannot be used to create two different GEMs, because the JSON copy removes cycles and performs a deep copy of the initial shared object graph. Second, GEMs can be used as a "drop in" replacement in message-based applications, as exchanging a GEM or a JSON message between two workers has the exact same semantics. Note that we chose JSON copy semantics also to avoid introducing new semantics that JavaScript developers would need to understand when using GEMs.

Creating a GEM with an initial object graph to be shared is optional, as performing a JSON copy of an object graph before transferring it as a message can be sometimes expensive. GEMs can also support a different pattern that does not require JSON-copying the object. Consider the following example of a GEM:

```
// import the gem from a module
var ownedGem = require('owned-gem');
// create and populate the gem with values
var gem = ownedGem.create();
// writes allowed before the gem is sent
for (var i in someValues)
    gem[i] = someFunctionOf(someValues);
// share the gem with some workers
for (var w in workers)
    workers[w].send(gem);
// once shared, the gem becomes read-only
gem.some++; // throws an exception!
```

In this example, the GEM is created with an empty initial state, which is populated by the GEM owner as fields are added. The GEM owner has the right to alter the GEM state as long as the GEM is not shared with any workers. In this way, the runtime overhead of JSON-copying a GEM from an existing object into a GEM-private memory space can be avoided in an efficient way, as the GEM-private object graph is built incrementally. In Section 5 the implementation of this as well as other GEMs will be provided, describing how the transfer mechanism is encoded in the GEM configuration object.

4.2 GEM Configuration Object

A GEM can be considered a safe container mediating accesses from any worker to the shared object graph it protects. The semantics of the GEM under concurrent access is specified in the configuration object used to create it. Different configuration objects can encapsulate different semantics, and therefore enable different parallel programming models. Examples of configurations are read-only access, multiple-readers/single-writer access, or memory partitioning. As the example in Section 3 suggests, GEMs can be used to implement even more advanced forms of data sharing, which we discuss in Section 5.

The high-level architecture of a GEM configuration is depicted in Figure 3, while a detailed overview of its components is provided in Figure 4. At its core, a GEM configuration is an object containing the following components:

(1) GEM public API: a GEM-specific API that is accessible to all the Node.js workers receiving the GEM.

(2) GEM meta API: a customizable metaobject protocol API [33] used by all the workers using the GEM.

(3) GEM state: global state accessible to all GEMs in all workers, and local state private to the worker that receives

GEM Public API	*Any arbitrary function that can be called by workers that received the* GEM. *An example is the* `atomic` *function used by Atomic* GEMs.
GEM Meta API	*Meta-object-protocol handling the following events:*
onGet	Trap called for every property `get` operation.
onSet	Trap called for every property `set` operation.
onCreate	Trap called every time a new instance of the GEM is created.
onFirstImport	Trap called the very first time a GEM of a given type is imported as a module.
onSend	Trap called before a GEM is sent from one worker to another.
enum, define, ...	Other traps used to access properties in the object graph.
GEM State	*Every* GEM *has access to some internal state that can be used to specify how concurrency is handled.*
Local	State that is private to a single GEM instance.
Shared	State that is specific to a GEM instance and is shared between all the workers that have access to the GEM (including the GEM shared object graph).
Static	State that is allocated at the moment the GEM is imported. It can be both worker-local and shared, and can be accessed by all the GEMs in the system.

Figure 4. Overview of the GEM API and state.

a GEM. State is private, and can be accessed *only* by the GEM public and meta API.

By combining these three components, a GEM can enable multiple forms of data sharing. Since GEMs are exposed to workers via message passing, a crucial aspect concerning their semantics regards the data involved in transferring a GEM from its owner to a receiving worker. After its creation, transferring a GEM operates as follows:

- The GEM owner creates a new GEM instance and sends a reference of it to the receiving worker.
- The new GEM instance holds a private reference to the shared object graph. The reference is *not* accessible to the worker user code, and can be accessed only by the GEM (meta and public) API.
- The new GEM instance also holds a local state. The value of the local state is specified in the GEM configuration.

Given this model, sending a GEM only has the cost of an object allocation and of a reference transfer. In Figure 5 and 6, the configuration objects of two example GEMs are described. The GEMs provide support for different forms of concurrent access to workers, and make use of all the features described before. In particular, the GEM in Figure 5 presents the implementation of a read-only GEM, while Figure 6 corresponds to the implementation of a *partitioned* GEM, i. e., a GEM that grants exclusive read and write access to disjoint ranges of a shared array to multiple workers in parallel. As the synchronization logic of both GEMs is not directly exposed, they can be used safely by workers without any risk of data races. In the following sections, we detail the internals of GEMs using Figure 5 and 6 as driving examples.

4.2.1 GEM Private State

The shared object graph used to create a GEM is always accessible from the GEM public API and from the meta API by accessing the `this.shared` object. As an example, this is done in Figure 6 at Lines 32 and 41, where its content is accessed by the workers. GEMs also provide an additional mechanism to access a shared utility state that is different from the shared object graph itself. This can be done by using the `internal` built-in property. An example usage of this property is to store some metadata that needs to be shared between workers. In Figure 6 this is used to share a counter keeping track of the ranges already assigned to workers. Finally, GEMs also provide an additional type of internal state, called `static`, which is allocated only once, when the GEM is imported into the application for the first time. The motivation for this type of GEM state (not shown in the figure) is to enable worker-local state that survives the allocation of a single GEM instance. Static state can be used in several ways, for example to keep track of all the GEMs used by a worker. The shared object graph as well as the object stored in the `internal` built-in property are *not* exposed to the worker, which cannot obtain access to them not even using reflection.

GEM-local state can be specified using a property named `local`. The property can be used by the GEM worker to model worker-local state. An example usage is depicted in Figure 6, where local state is used to keep track of the ranges in which a worker is granted access. When a partition is acquired, the range is checked to prevent unauthorized accesses to the shared object.

4.2.2 GEM Public API

Any GEM can have an optional public API to expose high-level programming models to workers. Such API implementations have the following characteristics:

- Every function defined in the API has *direct* access to the GEM shared object graph as well as to the GEM state. Access is granted by the GEM runtime via specific built-in objects (e. g., `this.shared`).
- Every function has access to a *private* built-in module called `Sync`, which provides concurrency control primitives, e. g., atomic *compare and swap* (CAS) operations.

The public API can use the `Sync` object to build a concurrency control mechanism for the GEM. For example, Figure 6 defines the `getRange` API. Similarly to the GEM state, the public API is declared by initializing the `public` property with an object that provides the API functions. In the example, the `getRange` function is exposed to every worker

ReadOnly GEM *configuration*

```
var readOnlyGem = {
  meta: {
    onGet : function(property) {
      // All reads are allowed
      return this.shared[property];
    },
    onSet : function(property, value) {
      throw "Read-Only access violation";
    }
}}
```

Worker usage

```
// Gems received: 'input' is read-only,
// and 'result' is partitioned
worker.on('message',
          function(input, result) {
  // Access read-only and part. gems
  result.getRange(function(from, to) {
    for (var r = from; r < to; r++)
      result[r] = someFunctionOf(
                    input[r], result[r]);
  });
  // The onSet meta API prevents
  // accesses outside of "getRange"
  result[42] = 42; // throws an except.
});
```

Figure 5. A GEM implementing read-only access for multiple workers to a shared object, and a worker receiving a ReadOnly GEM and a Partitioned GEM for parallel processing.

that has access to the GEM. `getRange` takes a lambda function as argument, which is executed atomically. When executed, the lambda function has full read and write access to the given range of indexes on the shared object graph. Internally, the `Sync` object is used to implement a thread-safe atomic counter that is acquired before executing the function.[1]

The GEM public API model enforces encapsulation and protection: workers can neither access the shared object graph nor the `Sync` object unless they are explicitly exposed by the public API of an "unsafe" GEM.

Having access to the `Sync` and the shared objects enables the implementation of custom synchronization policies. By using only the public API, a GEM can implement any form of shared-memory concurrent data structure. It is the responsibility of the GEM developer to decide which level of safety and consistency is to be exposed to users. In our implementation, the `Sync` object provides the concurrency semantics of the Java Memory Model [38]. This means, memory effects can be reasoned about with *happens-before* relationships. The `Sync` object implements the following low-level facilities:

[1] The example is simplified for brevity.

Partitioned GEM *configuration*

```
var partitionedDynGem = {
  internal: {
    // an atomic counter
    idxCounter : Sync.newAtomicLong(0),
  },
  // index range for which the worker
  // has exclusive access
  local: { range : {from:-1, to:-1} },
  public: {
    getRange: function(lambda) {
      var idx = this.internal
        .idxCounter
          .atomicIncrement(range);
      this.local.from = idx;
      this.local.to   = idx+range;
      lambda(from, to);
      this.local.to   = -1;
      this.local.from = -1;
    }
  },
  meta: {
    onCreate : function(shared) {
      if (!isTypedArray(shared))
        throw "this gem supports "
          + "only index-based arrays"
    }
    onGet : function(property) {
      if (parseInt(property) >=
            this.local.from
          && parseInt(property) <
            this.local.to) {
        return this.shared[property];
      } else
        throw "Out-of-bound access";
    },
    onSet : function(property, value) {
      if (parseInt(property) >=
            this.local.from
          && parseInt(property) <
            this.local.to) {
        this.shared[property] = value;
      } else
        throw "Out-of-bound access!";
    }
}}};
```

Figure 6. A GEM implementing exclusive access for multiple workers to array partitions.

- Atomic operations: *fetchAndAdd*, *compareAndSet*. Common atomic operations implemented in hardware, as well as access to thread-local storage.
- Concurrent access: *load*, *store*, and *has*. Primitive operations enforcing a happens-before relation between accesses on the shared object. They correspond to *volatile* operations, according to the Java Memory Model (JMM)

The choice of the Java Memory Model is arbitrary, and based on our choice to implement GEMs in an engine on top of the Java Virtual Machine (JVM). In principle, it would be possible to extend the GEMs model to support other memory models and operations. This would affect the number and the type of GEMs that could be developed, but not the GEMs model itself. Using the JMM for GEMs is further discussed in Section 6, where we highlight the main consequences of such design choice.

4.2.3 GEM Meta API

GEMs can define a metaobject protocol [33] through a custom meta API, to implement fine-grained access and concurrency control at the level of a single object property, preventing unauthorized accesses. Figure 5 depicts the `onGet` and `onSet` functions as an example of the meta API usage. In the *worker usage* part of the figure, the meta API is used to specify *implicit* custom operations to be executed by the GEM upon property read and write accesses. This ensures that all the properties of the object are accessed only in combination with the `getRange` API. Other attempts to access the GEM correspond to a misuse of the `getRange` API, and result in an exception (Line 12). Besides the two functions `onGet` and `onSet`, other meta functions (also called traps [47]) to intercept any form of property access to the shared object graph can be specified, e. g., to intercept a property delete operation. Furthermore, the model supports traps related to the lifetime of a GEM such as `onCreate`, which is called when the GEM is first allocated, and `onSend`, which is executed before a GEM is transferred from one worker to another one. A summary of all the possible meta functions supported by the model is depicted in Figure 4. Functions defined in the meta API have the same properties of functions defined in the public API. In particular, they can access the shared object graph via `this.shared` and they have access to the low-level `Sync` built-in.

4.3 Thread Safety and GEMs Programming

The runtime semantics of GEMs is specified using a low-level API that can be used to expose arbitrary parallel programming models to workers. Such API (e. g., the `Sync` and `shared` objects) is private, and workers receiving a GEM cannot directly access unsafe primitives such as, e.g., CAS operations or locks. Despite being thread-safety the main motivation for GEMs, the GEM model does not explicitly prevent the creation of arbitrary unsafe GEMs. The reason for this design choice is that some power users might still need to use unsafe GEMs for certain tasks (e.g., to perform low-overhead logging of events without strict consistency requirements). We rely on the Node.js community to create and design new GEMs which could introduce novel programming models that could benefit specific Node.js applications.

As discussed earlier, the internal GEM programming model requires a deep understanding of concurrency. Thus, we expect the roles of GEM developers and GEM users to be different. GEM users will typically import existing GEMs into their applications as they need to share state between workers in a safe way, but they will rarely develop GEMs themselves. To emphasize this distinction, the GEMs implementation is designed so that new GEMs can only be exposed to Node.js applications in the form of external Node.js modules. A Node.js user cannot directly create new types of GEMs. This is analogous to how the Node.js ecosystem supports native extensions using languages other than JavaScript (e. g., C++): such extensions cannot be explicitly accessed by Node.js applications, and needs to be exposed to JavaScript using a pre-packed module. In the case of GEMs, this means that the configuration object and API of a GEM can be specified only from within Node.js modules, and not by Node.js applications using the GEM. To enforce such model, we rely on the existing Node.js package management system for the distribution of GEMs, and we ensure that a GEM's semantics cannot be altered after the GEM is imported into an application. In this way, any Node.js user can import existing GEMs in their application without having to create themselves a new type of GEMs when they want to share some application-specific data between workers. Ideally, Node.js developers should just import existing GEMs into their applications in the way Java developers use libraries such as `java.util.concurrent`.

GEM name	Description
ReadOnly	Every worker has read-only access to all elements of the shared object graph. Write attempts cause an exception.
Owned	A worker has exclusive access to all elements of the shared object graph. Concurrent accesses by other workers cause an exception.
Partitioned	Workers have read/write access to disjoint subsets of elements of the shared object graph. Attempts to read or write outside of the partition cause an exception. The partition is assigned statically.
Partitioned-dyn	Same as **Partitioned**, but the partition assigned to each worker is defined dynamically by the GEM, e. g., to enable load balancing.
Atomic-LK	Workers have read/write access to every element of the shared object graph. The GEM provides an API for thread-safe access implemented with a lock. Attempts to access the shared object graph without owning the corresponding lock cause an exception.
Atomic-STM	Same as **Atomic-LK**, but the GEM enables concurrent access using an STM.

Figure 7. Overview of the GEMs discussed in Section 5.

5. Parallel Programming with GEMs

This section discusses examples of GEMs enabling safe programming models. Some of these GEMs are designed

to specifically address server-side workloads. An overview is provided in Figure 7.

5.1 ReadOnly and Owned GEMs

Since fully-isolated message passing is the default parallel programming model in Node.js, the first GEM extending it provides parallel read-only access to shared data for multiple workers. The GEM enforces at runtime that workers accessing it can only perform read operations on the protected object. When a worker attempts to write to the shared data, the GEM throws an exception instead. We call this a **ReadOnly** GEM. An example of its usage and of its internal structure has been discussed in Section 3 and in Figure 1. The GEM relies only on the meta API for intercepting read and write operations, and does not expose any additional public API. From the worker's perspective, the GEM behaves like a normal JavaScript object that has been received via message passing. However the worker has no direct access to the underlying object, as it only received a GEM. Thanks to the JSON copy mechanism used to build the object graph of the GEM, this enables to optimize existing message-based applications just by employing read-only GEMs rather than normal messages. This is possible as long as the object received (resp. sent) by a worker is only read. This scenario is common for message-passing applications, as in many parallel applications the sender of a message does not modify it afterwards, and the receiver usually reacts to the received message by performing some computations and subsequently generates a new message.

For the case that write access is necessary, we can introduce another type of GEM—similar to the ReadOnly one—called **Owned** GEM. Such type of GEM can be used to enable read and write access to a shared object for a single worker at a time. In other words, the GEM can be used to implement a mechanism of ownership delegation to ensure that while a worker is operating on it, no other workers can access it. This type of GEM is implemented using only the meta API, by performing a single CAS operation to *acquire* the ownership on an object on the first access, and by registering in the GEM-local state a pointer to the current thread owning the GEM. The following is an example of the meta API for property writes:

```
function onSet(property, value) {
  if (this.local.owner !=
            Sync.currentThread())
    throw "Cannot access this gem "
          + "from another thread.";
   this.shared[property] = value;
}
```

5.2 Partitioned GEMs

The ReadOnly and Owned GEMs supplement message passing by enabling more data parallel applications on Node.js. However, they do not enable arbitrary "read-after-write" parallel access, and therefore limit the class of parallel applications that they could support. To cover more applications, the **Partitioned** GEM introduced in Section 3 can be used. This GEM can be implemented using policies and APIs different from the ones discussed in Section 3. For example, the partitioning can be either static (i. e., done at GEM creation time) or dynamic (i. e., implemented by the GEM itself). This kind of GEM can also be used to build programming models in the form of global partitioned address space [19], or more generically can be used to implement scatter/gather computations [17].

5.3 Atomic GEMs

While Partitioned GEMs enable safe shared-memory parallelism, it is not always possible to define disjoint partitions, so that some computations remain better expressed by other abstractions. For these computations, we can introduce the **Atomic** GEM which enables concurrent read and write access to all elements of the shared object graph. To ensure thread-safety, the GEM provides a public API that workers must use for concurrent accesses. Misuse of the API results in an exception. The following is an example usage of the GEM:

```
// Receive the gem from somewhere
var gem = ...; var index = 42;
// get read/write access on it
var value = gem.atomic(function() {
  // --- Transaction begin
  if (gem[index]  // log Gem read
        == undefined) {
    // gem write: add to 'redo' log
    gem[index] = new Value();
  }
  return gem[index]; // log gem read
  // --- Transaction end.
  // --- Retry if aborted.
});
// access from outside of the
// 'atomic' block is forbidden
// and throws exception
var wrongAccess = gem[index];
```

The GEM provides the `atomic` function as its explicit API, which can be called to obtain safe access to the shared object graph. Once called, operations on elements of the GEM can be performed safely. Furthermore, read or write accesses are only possible through the `atomic` function. The meta API is used by the GEM to throw an exception when a worker attempts to perform a read or write operation outside of the `atomic` function call. The comments in the code correspond to the (implicit) invocations to a Software Transactional Memory runtime (STM) that is implemented to ensure atomicity for this GEM. Since the GEM throws an exception when accessed from outside of the transaction, the STM implementation can enforce *strong atomicity* [27], because non-transactional code cannot access the shared object

graph. STM is only one of the possible internal implementations for this GEM. For our evaluation we have implemented it using the TLRW STM algorithm [22], as well as a lock-based implementation (called **Atomic-LK**). The STM implementation is based on a lightweight usage of reader/writer locks, while the lock-based implementation is based on in-order acquisition of locks. A comparison of the scalability and performance of these two approaches is out of the scope of this paper.

6. Implementation in Graal.js

The GEMs model requires a Node.js engine with multi-threading support to enable multiple isolated workers to allocate shared objects in a common memory space. Since the Node.js runtime does not provide a common memory space, as discussed in Section 2, we based our implementation on a modified version of the Graal.js engine [49], a fully-compliant ECMA6 implementation of Node.js running on the JVM. Being based on the JVM, Graal.js can spawn independent workers as Java threads, leveraging the existing Java heap as common memory space. JavaScript workers still guarantee isolation however, although each JavaScript object is allocated in the JVM heap. Implementing the GEMs model using the V8 engine (used in Node.js) would also be possible, but would require modifications to the garbage collector of the engine, which does not support concurrent allocation of objects from multiple "Isolates" (using V8's terminology).

Graal.js is a state-of-the-art JavaScript execution engine based on a self-optimizing AST interpreter that is compiled to highly optimized machine code via partial evaluation of the AST nodes performed by the Graal [49] just-in-time compiler.

We modified the Graal.js AST interpreter to support GEMs' meta API. The JSON copy of the (optional) initial object graph of a GEM is implemented using an algorithm with the semantics of JSON encoding and decoding.[2] After the object graph has been copied, each of the objects in the new graph that does not correspond to a primitive value (i. e., objects and arrays literals) is *wrapped* with Proxy objects [47] that enforce the GEM meta API. This makes it possible to ensure that all objects in a newly-created graph will be accessed using the GEM API. As the full object graph of the shared object is JSON-copied when a GEM is created (and the resulting copy is private to the GEM), it is not possible to have two object instances referenced by two distinct GEMs at the moment the GEM is created.

Compared to ECMA6 Proxy objects, the Proxy objects used to implement the GEM meta API have a simpler semantics. They provide only *onGet*, *onSet*, and other common access traps (e.g., *has*), whereas ECMA6 proxies feature a richer set of traps that can be used to model other JavaScript-specific aspects such as object creation and extensibility. The reason for this design is that ECMA6 proxy objects can be used to model the interaction with *any* JavaScript object, whereas GEM traps model only the interaction (i.e., reading and writing of properties) with the GEM shared object graph, which has the JSON semantics described before. A second reason for such design is that GEMs do not need to support the full semantics of ECMA6 proxies (which is very rich and can be used to express complex meta-object protocols). For example, the ECMA6 standard specifies that the return value of a *property get* trap is checked against the property descriptor of the object handler (if any) to ensure that the value returned by the trap invocation is compatible with the value specified in the object's property descriptor [6]. Supporting such semantics for GEM objects is not needed, because GEM shared objects do not have property descriptors, since they are JSON object graphs.

The implementation of the GEMs meta API in Graal.js is based on a common basic proxy implementation that is shared between ECMA6 proxies and GEMs. The implementation of ECMA6 proxies adds to such common implementation the additional needed runtime semantics, which is not needed for GEMs. The proxy objects used for GEMs have the following characteristics:

- Their traps never change. Once a GEM is created, it is not possible to change its set/get traps anymore.
- The traps encode the full semantics of the meta-object protocol that a specific GEM is implementing, and no extra checks have to be performed on other aspects of the shared object protected by the GEM nor on the return value of a GEM invocation.

By sharing the basic proxy mechanisms, the runtime support and the optimizations by Graal.js for proxy objects can be leveraged by GEMs, too. In addition, the engine applies the following two optimizations utilizing the specific characteristics of GEMs:

- Since GEM traps for reading or setting a property never change, they can be inlined very aggressively for each property access operation.
- Since the traps encode the entire semantics of the meta protocol of a GEM, no extra runtime checks have to be performed on the values returned after the trap execution. In other words, they can be considered like any other JavaScript function, with the additional advantage that they are monomorphic, as their body never changes. This enables all optimizations performed by the Graal.js runtime to be performed on GEMs as well, transparently.

Graal.js modifications. Overall, the modifications required to support GEMs in Graal.js were very localized, as their design enables GEMs to benefit from most of the optimiza-

[2] The object is created with a copying algorithm that is semantically equivalent to the combined call `JSON.parse(JSON.stringify(object))` on the object to be shared, and corresponds to the normal way objects are exchanged between workers in Node.js.

tions that a modern JavaScript engine already performs for property accesses (e. g., polymorphic inline caching [29]) and function calls. For example, because the GEM traps are constant, inline caches always cache the GEM trap function at each property access. As a result, the Graal compiler assumes most of the GEM traps to be constant for the monomorphic case, and performs very aggressive inlining, thus enabling other optimizations (e.g., partial escape analysis) for the entire compilation unit (e.g., a function using the GEM). Such optimizations are not specific to the GEMs model, but are required to guarantee the minimal runtime overhead of GEMs (cf. Section 7). Without such optimizations, the GEMs model would be impractical, as the overhead for accessing object properties would be too high. Another consequence of the GEMs design in combination with such optimizations is that the impact on the compiled code size depends almost entirely on the size of the GEM traps: for the ideal case (i.e., when the trap function only reads/writes a property), the machine code produced by Graal is close to the one of a normal property lookup, since Graal can remove all the function calls and temporary allocations. Of course, the overhead may grow for GEMs with complex internal logic.

Supporting the Java Memory Model. When a GEM is accessed concurrently by two or more Node.js workers, a definition of an *happens-before* relationship is needed. Since our implementation is based on the Graal.js engine (that is, a Java-based JavaScript engine) we rely on the Java Memory Model to model the semantics of concurrent accesses to the GEMs' shared object graph. JavaScript applications which do not use GEMs are not affected by this design at all.

The main consequence of this design choice is that accesses to properties of the GEM shared object without synchronization have the same undefined observable behavior of concurrent non-volatile accesses to Java fields. Conversely, accessing the object graph using any synchronization primitive (e.g., incrementing a volatile value atomically) has the same guarantees that the same synchronization primitive would provide in a Java application. The way the JMM properties are exposed to the final JavaScript developer depends on how the GEM exposes and permits accesses to its shared graph. For GEMs providing safe access to their shared graph, the presence of the JMM should be completely transparent to the final user, as the GEM should ideally prevent concurrent non-synchronized access, and should encapsulate any synchronization primitives without explicitly exposing them.

7. Evaluation

To evaluate the performance benefits of GEMs, we implemented the GEMs discussed in Section 5 to run on the Graal.js engine. For each GEM, we developed benchmarks to assess their performance. For each benchmark we also implemented an equivalent version in pure Node.js. We report the results for each benchmark without GEMs on Node.js and Graal.js. Furthermore, we report the results for each benchmark using GEMs with Graal.js. Since our GEMs implementation is based on Graal.js, there are no numbers for Node.js using GEMs (cf. Section 6). The benchmark versions are executed with the same input workload, e. g., workload generator, input data, or files.

The experiments have been run on a server-class machine running Ubuntu 14.04 equipped with 128GB of RAM and two 8-core Intel(R) Xeon(R) CPU E5-2680 @ 2.70GHz, which correspond to 32 hyper-threads. Each CPU has 2MB of L2 cache and 20MB of L3 cache. We used Node.js version 6.0 and Graal.js version 0.13. The Web applications use the Wrk (v4.0) HTTP workload generator. An overview of all the benchmarks along with the GEMs they use is depicted in Figure 8. Our main goal is to show that GEMs benefit from the usage of shared memory. The numbers shown correspond to average performance data obtained using the Kalibera performance evaluation methodology [31]. Standard deviation is below 10%.

7.1 Meta API Overhead

A crucial aspect of the GEMs model is the meta API, since an unacceptable overhead for accessing GEM properties would nullify the benefits of shared memory. To assess the overhead of the meta API, we use microbenchmarks to stress the `onGet` and `onSet` meta API. The benchmarks perform an increasing number of read and/or write operations on a GEM. We compare the GEM performance taking as a baseline the standard JavaScript property read and write opera-

Benchmark	Description & GEM used for the implementation.
Ping	One-to-one message latency (Owned GEM).
Ring	One-to-many data distribution (Owned GEM).
Multicast	One-to-many throughput (ReadOnly GEM)
Dispatch	Many-to-many message latency (ReadOnly GEM).
Access	GEM access performance (ReadOnly GEM).
Scale-Up	Stateless HTTP requests serving with internal communication between workers (ReadOnly GEM).
Immutable	HTTP requests serving with shared (GEM) or replicated (Node.js) immutable data (ReadOnly GEM).
Cache	HTTP requests serving with high read contention on shared consistent data (Atomic-STM GEM).
Wordcount	Parallel calculation of the distribution of words in a data structure (ReadOnly & Partitioned GEMs).
Grep	Parallel scraping of a pattern in a sequence of tokens (ReadOnly & Atomic-LK GEMs).
SHA	A parallel calculation of the SHA of a big document (Partitioned-dyn GEM).
Crc32	A parallel calculation of a CRC code (ReadOnly & Partitioned GEM).
Primes	Parallel primes number calculator (Partitioned GEM).
Mandelbrot	Parallel Mandelbrot set calculation. (ReadOnly & Partitioned GEM).

Figure 8. Overview of the benchmarks used in Section 7.

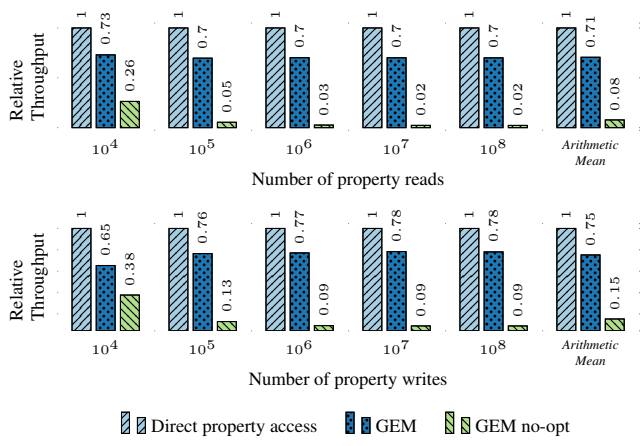

Figure 9. Micro-benchmarks measuring the impact of the meta API on the performance of property accesses

tion, i. e., not using GEMs. Figure 9 summarizes the results of the experiments reporting the slowdown with respect to the baseline (higher is better).

As the graphs show, reading or writing from a GEM property using the meta API (i. e., GEM) has limited slowdown compared to the pure JavaScript operation (i. e., direct property access). The overhead (less than 30% on average, see last column) makes it possible to use GEMs as a *drop-in replacement* of send-by-copy messages, as we will show in the rest of our evaluation. The optimizations performed by the Graal.js engine are essential to make the approach practical. To assess the impact of such optimizations, we have disabled inlining and polymorphic inline caching for the microbenchmarks. When all optimizations are disabled (i. e., GEM no-opt) accessing a GEM is one order of magnitude slower than accessing a normal object. Such overhead would of course make the GEM model less attractive as it would limit the type of applications where GEMs can be used.

7.2 Message Passing Using GEMs

GEMs can be used in common message-passing applications as a replacement for messages when shared memory is available. For example, ReadOnly GEMs can be used to implement multicast (i. e., one-to-many) operations, while Owned GEMs can be used to send data from one worker to another via ownership transfer.

We developed a selection of benchmarks inspired by common Actor benchmarks [30] used to measure the communication overhead of workers-like systems. Each benchmark consists of some workers sending some objects for a fixed number of times to other workers. All the benchmarks transfer objects between multiple workers. The first four benchmarks in Figure 10 receive an object and send it back to the sender or to other workers. Each benchmark differs in the communication topology and the type of GEM used:

- *Ping* is a benchmark where two workers exchange an object for a fixed number of times (10^3). The performance is measured for objects with an increasing size, and the performance of plain Node.js is compared against an implementation using an owned GEM.
- *Ring* is a benchmark where a "master" worker exchanges an object with 16 parallel workers. The master worker acts as a network proxy, and sends the object to each worker sequentially, that is, it waits for a worker to have received the object before sending a message to the worker in the ring. The operation is repeated for a fixed number of iterations (10^3), with an increasing object size. An owned GEM is used in the GEM-based implementation of the benchmark.
- *Multicast* is a benchmark where a "master" worker sends an object to 16 workers in parallel, i.e., without waiting for each worker's reply. The multicast operation is repeated for a fixed number of iterations (10^3), and a readonly GEM is used so to share the same object with all the workers.
- *Dispatch* is a benchmark where 16 workers exchange messages based on message content. Each worker generates a message and sends it to a random receiver. After receiving it, the worker re-sends the message to another random worker. The operation is repeated for a fixed number of messages (10^4), and a readonly GEM is used for messages.
- *Access* is a benchmark where two workers exchange an object of fixed size (10^4 bytes). After the object is received, the worker reads some of its properties before replying. A readonly GEM is used, and the operation is repeated for a fixed number of iterations (10^2). The size of the object exchanged between workers is constant, while the number of properties read increases between executions.

We compare GEM-based executions to equivalent implementations with Graal.js and pure Node.js.

As Figure 10 shows, the cost of exchanging a GEM between workers is considerably lower also for objects of small size. This is expected, as exchanging a GEM corresponds to an in-memory transfer, whereas transferring even a small object between two isolated workers requires JSON serialization, inter-process communication, and data de-serialization into the memory space of the receiver worker. As the size of the objects transferred between workers grows, both Node.js' and Graal.js' performance start to degrade. This is also expected, and it is due to the cost of transferring data between the memory spaces of the workers. Using GEMs, the size of the objects transferred does not affect the performance of the benchmarks. These results confirm that GEMs can be a valid alternative to standard messages.

The last benchmark (Access) is similar to the first one (Ping), but the object exchanged between workers has a fixed

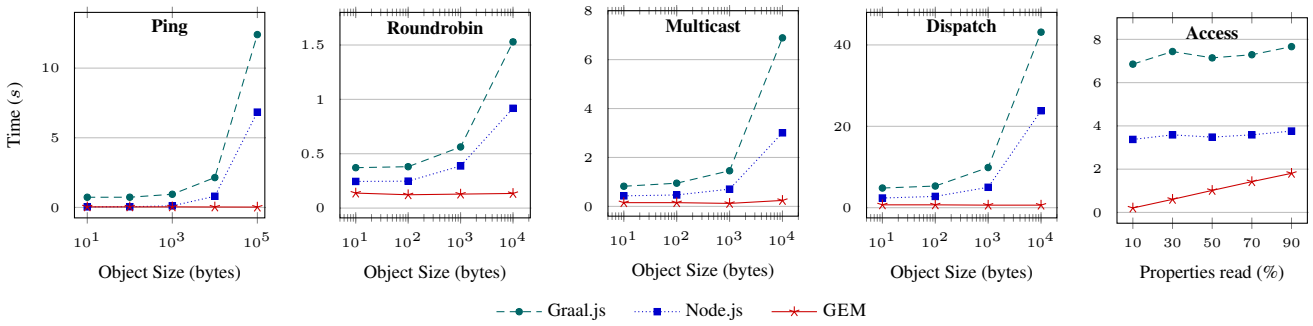

Figure 10. GEMs message passing performance. In all the considered workloads, Node.js performance degrade as the message size increases. Graal.js has comparable performance to Node.js and similar linear dependency to the message size. When GEMs are used, the size of the message does not affect performance, as in-memory transfer is used. The only exception to this is represented by the access micro-benchmark.

size. Instead of the object size, the benchmarks changes in the number of elements that each worker reads from the incoming message. In this case, the performance of the GEM-based implementation are still better than the ones of Graal.js, but the number of operations performed on the GEM affects the overall performance of the application. This result is expected, and shows that GEMs can reduce the communication overhead even when the entire message is used by the receiver. This also suggests that ReadOnly and Owned GEMs are particularly useful when a receiver worker does not *need* the entire message, but only a fraction of it.

7.3 Node.js Applications

One common usage of workers in Node.js applications is to *scale-up* Web services. In such applications, workers accept incoming requests (from independent clients) in parallel, perform some computation, and reply. Recently, the so-called Microservices [48] architecture has popularized the usage of workers in this way. When the service requires some notion of state, the share-nothing model of workers requires developers to use external services such as Memcached [1] or Redis [2] to share data between workers. Such services offer distributed storage and failure tolerance, but are often used merely to *enable* temporary shared state within a single multicore machine. This usage has become so common that services such as Redis provide synchronization (e. g., using locks) and atomicity (using a form of software transactions) on the shared memory they expose to workers. Such approach, however, comes at the cost of increased service latency, as it requires each worker to access an object cached in the memory space of the external service, to transfer it into its memory space (usually using TCP connections) and to re-materialize it into its heap space before generating the reply for a client. When such cost is not acceptable (e.g., for latency-bound services) workers avoid using external caching systems, and rely on per-process local caching systems (e.g., TTL [9]). Using such a solution, each worker has its private cache, and the web service trades performance (latency) for memory consumption.

GEMs can be used in such applications to avoid using external services when they can be replaced with shared memory, avoiding at the same time the cost of replicating the cached data into each worker's memory space. In the third part of our evaluation we focus on three types of Web applications that rely on some notion of application-level state, namely stateless, immutable, and caching. We developed a Web application using GEMs for each type, and we compared them against an equivalent Node.js implementation (results are depicted in Figure 11):

- In the first application (*Scale up*), workers do not share any local state, and interact with each other to generate the user response. This scenario is common in the Microservices architecture, where some worker-private state is combined to generate some application-level state to process a client request. The application requires the interaction of two distinct workers for each client request: once a request arrives, it is received and then forwarded to a background worker for processing. By decoupling the workers that accept incoming requests from the ones that perform the computation, multiple requests can be accepted in parallel. Using GEMs to implement the messaging between workers, the application performs considerably better. This is expected, and it is due to the reduced overhead of data transfer.
- In the second application (*immutable*), workers own an immutable data structure (e. g., a registry of immutable user data) that they use to generate the client response. The application involves one worker per request and demonstrates the benefits of sharing data instead of replicating it. Each worker owns a private copy of a replicated in-memory data structure which is used to generate the response. In this case the Node.js and the GEM appli-

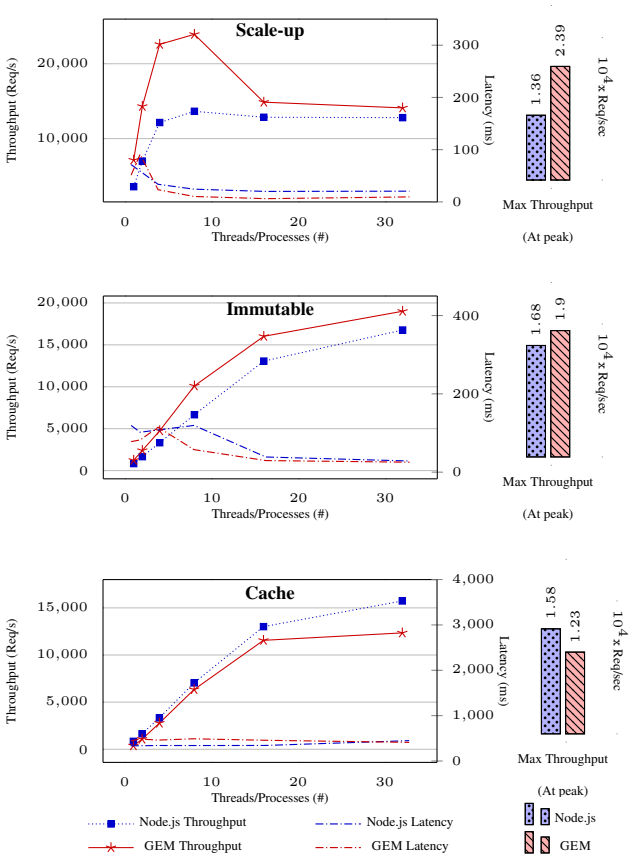

Figure 11. Web services benchmarks. Latency and throughput of pure Node.js applications compared against their equivalent GEM-based implementation. Node.js's scalability and peak performance are comparable with the ones of GEMs, which do not need data replication or external services.

cations scale identically. However, each Node.js worker has a *replicated* copy of the data structure, resulting in a larger memory footprint, while the version with GEMs shares a single copy directly. For big immutable data structures, the memory saving can be highly beneficial.

- In the third application (*cache*), workers access and modify an in-memory cache. The cache is mostly read, and less frequently modified. The application (*cache*) uses Node.js and Memcached to implement an in-memory data structure, and it is compared against a GEM-based implementation using an Atomic-STM GEM to perform the concurrent updates. Since most of the requests to the cache are cache hits, the transactional memory performs mostly read-only transactions. For this reason, the performance of the service are comparable to the ones of Memcached. The scalability curve is however different, as the service achieves its best throughput at 16 threads. This can be explained by observing that the machine used for the experiments has 16 physical cores. When hyper-threading is used (i. e., at 32 threads), the synchronization required by the STM runtime degrades the performance of the service. In absolute terms, however, GEMs can achieve comparable performance, without the additional architectural complexity of having to deploy two systems, and program against the external Memcached API.

7.4 Parallel Programming

GEMs can also be used to write parallel computing applications. To highlight the benefits of shared memory, we developed a selection of common data-intensive and CPU-intensive benchmarks using GEMs, and we compared them against their pure message-based equivalents in Node.js. Results are depicted in Figure 12. The first two benchmarks (Wordcount and Grep) are data-intensive, while the latter are more CPU-intensive:

- *Wordcount* and *Grep* are benchmarks that read from a shared file system some data and process it to either compute some word distribution or to search for some keywords. Both benchmarks require exchanging a relevant amount of data (i.e., the files) between workers, with a considerable impact on the overall application performance.
- The remaining benchmarks perform some computations based on an input data structure. *Primes* scans a list of random numbers counting the number of prime numbers in it; *Mandelbrot* computes the mandelbrot set from a matrix; *Sha* and *Crc* encode a long sequence of characters. In a pure message-based implementation each application can be implemented using some notion of workload distribution (e.g., scatter/gather [17] or MapReduce [20]). For some applications the cost of transferring data can have an impact on the overall computation. By using GEMs, most of the data-transfer-related overheads can be reduced.

As the figure shows, using GEMs for data-intensive computations improves the performance of the applications. Note that the impact of GEMs can be more evident when a limited number of cores is used. This happens because with a small number of workers the size of the messages to be exchanged increases, and so does the overhead for sending data to workers. This result suggests that implementing such applications using Node.js is *only* reasonable as long as communication is not the bottleneck. Conversely, applications using GEMs are not affected by this phenomena, as the size of the message is not relevant for the performance of the application. Considering CPU-intensive applications, Node.js and GEMs scale similarly, since data transfer is not a bottleneck for most of the applications. Nevertheless, the two implementations have a very different programming model, as using GEMs enables direct access to data, therefore resulting in a more compact definition of the algorithms.

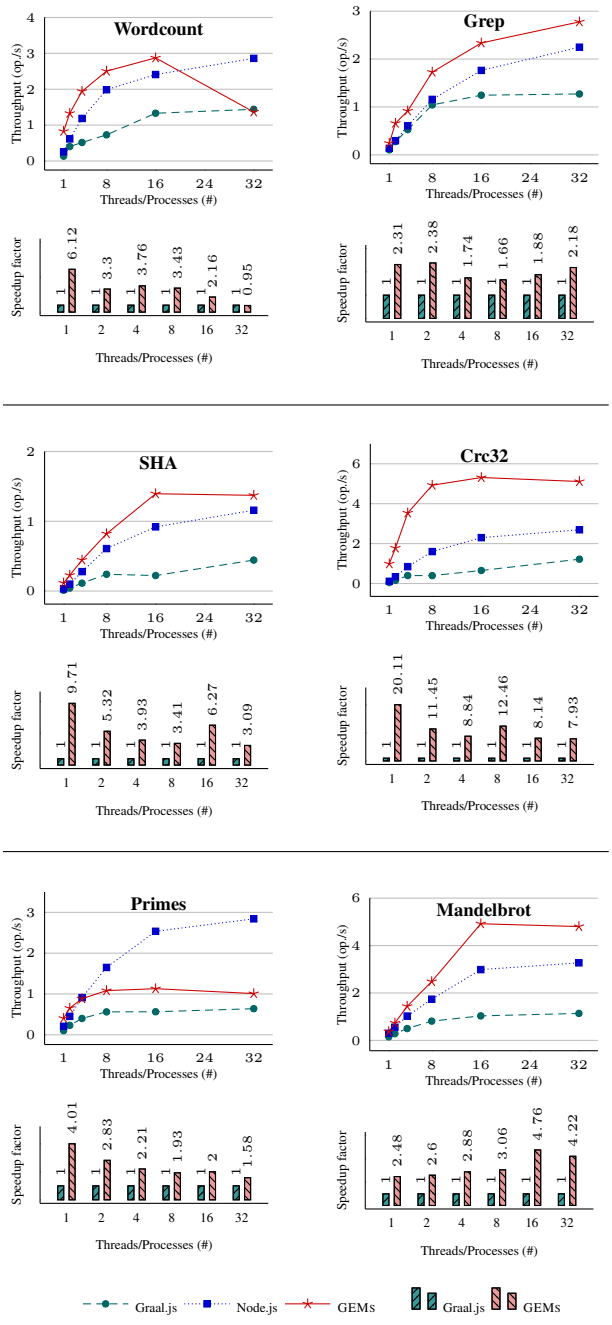

Figure 12. Parallel applications. The *throughput* measures the number of operations per second. The throughput of Node.js applications is affected by the size of the messages, while GEM-based applications are not. The *Speedup factor* shown is the speedup of GEMs with respect to Graal.js. The GEM-based implementation is consistently faster than its baseline.

8. Related Work

GEMs *and Object Capabilities.* The concepts of object capabilities originates from work of Dennis and Van Horn [21] and since then has been used in various ways in the context of programming models [40]. Some researchers have tried to define a set of useful capabilities for concurrent programming [13, 23]. A relevant example is the Pony [15] language, which features a type system that automatically enforces read or write access to shared objects between actors. Pony is itself inspired by the design of E [44]'s programming model, and shares some similarities with ownership-based models [12]. In contrast to such approaches, our technique works with an existing dynamically-typed language, and does not require any type system to enforce access rights to shared state. Moreover, the GEM model is more customizable than the one of languages such as Pony, as it allows GEM developers to implement complex sharing strategies that can rely on runtime informations only, e. g., to introduce *temporary* access rights (i. e., temporary ownership), as with the owned GEM we have described.

Parallelism for JavaScript. In the JavaScript ecosystem, related work includes WebWorkers [4], Cluster [8] and RiverTrail [28], which have been detailed in previous sections. RiverTrail has the notion of Temporal Immutability [39], which can also be emulated using a GEM. GEMs, however, are not explicitly targeted at data-parallel computations only. A proposal for so-called `SharedArrayBuffer` introduces a new typed data structure that can be shared between workers [10]. It introduces a raw binary buffer shared between workers, and could be implemented using a GEM, too. To our knowledge, other Node.js modules for parallel programming rely on share nothing parallelism or use external services to enable shared state.

Shared Memory Approaches for Non-shared Memory Systems. Going beyond JavaScript, other programming models for sharing state between isolated entities such as workers have been proposed [18, 42]. These approaches share with GEMs the goal of enabling shared state, but focus on specific use cases. GEMs however are a generic abstraction, which can potentially be used to implement such approaches. GEMs's meta API builds on the notion of Proxy [47] and metaobject protocols [33]. Unlike existing approaches, the meta API in GEMs is used to coordinate the access between multiple workers. Moreover, it works in combination with the notion of JSON copy previously introduced, and has access to the GEM-private `Sync` module for the implementation of concurrency control mechanisms. JavaScript itself supports proxies in the ECMA6 standard [6]. However, these proxies cannot be shared between workers and therefore cannot be used to implement a GEM.

Shared Memory Programming Models. The GEMs and their parallel programming models in this paper are inspired by models from other languages and frameworks. Delegation-based isolation, for example, is supported in different forms in actor-based models [37]. Partitioning is also

available in several models, e. g., all models based on partitioned global address space [19]. We do not claim novelty for the programming models enabled by GEMs, but we consider GEMs an innovation that enables the implementation of such programming models in shared-nothing environments. Furthermore, we see them as a mechanism that can be used to introduce sharing in a safe way and with very fine-grained control.

Approaches to introduce disciplined shared memory have a long tradition. One example are hyperobjects as introduced by Cilk++ [26]. They are programming abstractions that provide shared-memory between threads with a specific set of properties depending on the concrete hyperobjects, which is an approach similar to GEMs. Other models have attempted to combine shared memory and message-based models [18, 32]. In contrast to these approaches, GEMs do not force the adoption of a single specific model, and could be used to combine other models in higher-level structured forms (e. g., in the form of Skeletons [16]).

The zero-copy mechanism used for some GEMs (i. e., Owned GEMs) closely resembles zero-copy ownership-transfer, which is present in existing MPI frameworks, as the Ownership Transfer Interface library by Friedley et al. [25], or SOTER by Negara et al. [41]. The idea of ownership passing derives from previous systems, such as distributed shared memory (DSM) systems [43] and early cache coherence protocols [24]. An additional example of a system providing an extension for ownership passing is represented by the Generic Message Passing Framework [34, 35], which comprehends a message passing interface for C++.

9. Conclusion

In this paper we introduce Generic Messages (GEMs), a new abstraction to enable parallel programming in the context of WebWorkers-like models. GEMs are a generic form of messages that can be shared between workers to enable several forms of parallel programming models that rely or benefit from shared memory. Our evaluation shows that GEMs have a performance advantage for Node.js applications. The advantage comes from using shared memory to share state between workers in a thread-safe way.

In future work, we will investigate novel GEMs, with a special focus on high-level programming models for typical Node.js cloud deployments. Moreover, the GEMs model is applicable beyond Node.js, and we are investigating its application in other languages or language implementations with shared-nothing parallelism models. In doing so, the main open question is how to extend the GEMs model in order to support languages with semantics different from the one of Node.js.

Acknowledgment

Our research has been supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200021 153560). Stefan Marr was funded by a grant of the Austrian Science Fund (FWF), project number I2491-N31. We thank the VM Research Group at Oracle for their support. Oracle, Java, and HotSpot are trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

References

[1] MemCached Object Caching System. URL `http://www.memcached.org/`.

[2] Redis Data Structure Store. URL `http://www.redis.io/`.

[3] The Google V8 JavaScript engine. URL `https://code.google.com/p/v8/`.

[4] HTML5 WebWorker API. URL `http://dev.w3.org/html5/workers/`.

[5] Lambda: Microservices in the Cloud. URL `https://aws.amazon.com/lambda/`.

[6] ECMAScript Language Specification. v6. URL `http://www.ecma-international.org/ecma-262/6.0/`.

[7] The JavaScript Object Notation (JSON) Data Interchange Format. URL `https://tools.ietf.org/html/rfc7159`.

[8] Node.JS Cluster module, . URL `https://nodejs.org/api/cluster.html`.

[9] Node.JS TTL-cache: Simple in-memory object cache with TTL based per-item expiry, . URL `https://www.npmjs.com/package/ttl-cache`.

[10] SharedArrayBuffer Specification Draft. URL `https://www.chromestatus.com/feature/4570991992766464`.

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

[12] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proc. of OOPSLA '02*, pages 211–230, 2002.

[13] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proc. of ECOOP '01*, pages 2–27, 2001.

[14] Y. Cheng, A. Gupta, and A. R. Butt. An in-memory object caching framework with adaptive load balancing. In *Proc. of EuroSys '15*, pages 4:1–4:16, 2015.

[15] S. Clebsch, S. Drossopoulou, S. Blessing, and A. McNeil. Deny capabilities for safe, fast actors. In *Proc. of AGERE! '15*, pages 1–12, 2015.

[16] M. Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[17] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey. Scatter/gather: A cluster-based approach to browsing large document collections. In *Proc. of SIGIR '92*, pages 318–329, 1992.

[18] J. De Koster, S. Marr, T. D'Hondt, and T. Van Cutsem. Domains: safe sharing among actors. *Science of Computer Programming*, 98, Part 2:140–158, 2015.

[19] M. De Wael, S. Marr, B. De Fraine, T. Van Cutsem, and W. De Meuter. Partitioned Global Address Space Languages. *ACM Computing Surveys*, 47:62:1–62:27, 2015.

[20] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51 (1):107–113, 2008.

[21] J. B. Dennis and E. C. Van Horn. Programming Semantics for Multiprogrammed Computations. *Commun. ACM*, 9(3): 143–155, Mar. 1966.

[22] D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *Proc. of SPAA '10*, pages 284–293, 2010.

[23] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Proc. of ESOP '09*, pages 363–377, 2009.

[24] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics;(United States)*, 1, 1984.

[25] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership passing: Efficient distributed memory programming on multi-core systems. In *Proc. of PPoPP '13*, pages 177–186. ACM, 2013.

[26] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other cilk++ hyperobjects. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 79–90. ACM, 2009.

[27] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory*. Morgan and Claypool, 2nd edition, 2010.

[28] S. Herhut, R. L. Hudson, T. Shpeisman, and J. Sreeram. Parallel programming for the Web. In *Proc. of USENIX HotPar '12*, pages 1–6, 2012.

[29] U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-typed Object-oriented Languages with Polymorphic Inline Caches. In *Proc. of ECOOP '91*, pages 21–38, London, UK, 1991.

[30] S. M. Imam and V. Sarkar. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *Proc. of AGERE '14*, pages 67–80. ACM, 2014.

[31] T. Kalibera and R. Jones. Rigorous benchmarking in reasonable time. In *Proc. of ISMM '13*, 2013.

[32] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *Proc. of PPPJ '09*, pages 11–20. ACM, 2009.

[33] G. Kiczales, J. des Rivires, and D. G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, 1991.

[34] L.-Q. Lee and A. Lumsdaine. Generic programming for high performance scientific applications. In *Proc. of JCI '02*, pages 112–121, 2002.

[35] L.-Q. Lee and A. Lumsdaine. The generic message passing framework. In *Proc.of IPDPS '03*, pages 10 pp.–, April 2003.

[36] B. P. Lester. *The Art of Parallel Programming*. Prentice-Hall, 1993.

[37] R. Lublinerman, J. Zhao, Z. Budimlić, S. Chaudhuri, and V. Sarkar. *SIGPLAN Not.*, 46(10):885–902, Oct. 2011.

[38] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of PLDI '05*, pages 378–391, 2005.

[39] N. D. Matsakis. Parallel closures: a new twist on an old idea. In *Proc. of HotPar '12*, pages 1–6, 2012.

[40] M. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished. Technical report, Johns Hopkins University Systems Research Laboratory, 2003.

[41] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *Proc. of PPoPP '11*, pages 81–90. ACM, 2011.

[42] K. Palacz, J. Vitek, G. Czajkowski, and L. Daynas. Incommunicado: Efficient communication for isolates. In *Proc. of OOPSLA '02*, pages 262–274, 2002.

[43] J. Protic, M. Tomasevic, and V. Milutinović. *Distributed shared memory: Concepts and systems*, volume 21. John Wiley & Sons, 1998.

[44] J. E. Richardson, M. J. Carey, and D. T. Schuh. The design of the e programming language. *ACM Trans. Program. Lang. Syst.*, 15(3):494–534, July 1993.

[45] J. Shun. *Shared-Memory Parallelism Can Be Simple, Fast, and Scalable*. PhD thesis, 2015.

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

[47] T. Van Cutsem and M. S. Miller. Proxies: Design Principles for Robust Object-oriented Intercession APIs. In *Proc. of DLS '10*, pages 59–72, 2010.

[48] N. Viennot, M. Lécuyer, J. Bell, R. Geambasu, and J. Nieh. Synapse: A microservices architecture for heterogeneous-database web applications. In *Proc. of EuroSys '15*, pages 21:1–21:16, 2015.

[49] C. Wimmer and T. Würthinger. Truffle: a self-optimizing runtime system. In *Proc. of SPLASH '12*, pages 13–14, 2012.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proc. of NSDI '12*, pages 2–2, 2012.