Serialization Sets: A Dynamic Dependence-Based Parallel Execution Model

Matthew D. Allen
Srinath Sridharan
Gurindar S. Sohi

Technical Report #1644

August 2008

Serialization Sets:
A Dynamic Dependence-Based Parallel Execution Model

Matthew D. Allen
matthew@cs.wisc.edu

Srinath Sridharan
sridhara@cs.wisc.edu

Gurindar S. Sohi
sohi@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison

ABSTRACT

This paper proposes a new parallel execution model where programmers augment a sequential program with pieces of code called *serializers* that dynamically map computational operations into *serialization sets* of dependent operations. The runtime system executes operations in the same serialization set in program order, and may parallelize the execution of operations in different sets. For many types of applications, writing and debugging such programs is significantly easier than using existing parallel programming techniques, and results in deterministic parallel execution.

We describe the API and design of Prometheus, a C++ library that implements the serialization set abstraction through compile-time template instantiation and a runtime support library. We evaluate a set of parallel programs running on the x86_64 and SPARC-V9 ISAs and study their performance on multi-core, symmetric multiprocessor, and ccNUMA parallel machines. We find that parallel execution of programs written with serialization sets achieves performance comparable to traditional parallel execution models.

1. INTRODUCTION

As multicore processors become the default for mainstream computing, the search for parallel execution models has become one of the most important challenges facing the computer industry: models to achieve parallel execution of mainstream software must be developed. Meanwhile, the explosion of software applications has been enabled by modern programming practices that enable rapid construction and testing of software. These include programming in object-oriented languages such as C++, C# and Java, coupled with other concepts such as separate compilation, dynamic linking, and managed run-time systems. For the computer and information technology industry to continue to grow as it has in the past, it is critical that we find a solution that allows parallel execution of programs without compromising programming productivity. Such a solution should not hinder, and in fact even encourage, modern programming practices.

Almost all the solutions that are being explored by most of the research community build upon several decades of knowledge about parallel execution. These approaches can generically be described as follows: an entity (e.g., programmer or compiler) analyzes an application to determine *independence* among computations. Such independence is expressed *statically*, and when the independent computations interact (i.e., some sub-computations are *dependent*), they must be *synchronized* to ensure correct execution. An canonical example of this approach is the multithreading model which has, to date, been one of the most widely used models for expressing and achieving parallel execution in applications.

This template for parallel execution suffers from several difficult problems that are likely to place significant barriers for its widespread adoption. First and foremost, some entity must reason about the applications to determine which computations are independent. (The first step in doing so is determining which computations are, or may be, dependent.) Second, such independent computations are expressed statically and when they are dynamically executed, they execute in a non-deterministic manner. Third, when threads interact via operations on shared data, they must be carefully synchronized to ensure mutual exclusion. Incorrect synchronization can lead to data race errors, where the result of the computation depends on the arbitrary interleaving of thread operations. Fourth, employing threads admits other types of errors not present in sequential programs, including deadlock, livelock, and priority inversion. Lee argues that these problems, rooted in non-determinism, make the use of threads infeasible for many applications [11].

Considering the foremost challenge above, namely the analysis of the application, observe that much more information about dependence relationships amongst computations—the first step in determining independence—is available at run time. Thus one can conceive of an execution model where the "threads" of the canonical static multithreaded program can be determined by dynamically learning about independence (by first learning about dependence). If that could be done, then there may be no reason to also represent the threads statically: the static program will essentially look like a sequential program. And if the static representation leads to a deterministic execution, there is no need for synchronization constructs such as locks, and the problems due to non-determinism will disappear.

This paper proposes *serialization sets* (Section 2), a programming abstraction that retains much of the simplicity of sequential programming, and conveys dependence information to a runtime system to facilitate opportunistic parallelization of independent computations. Programmers specify dependence calculations by writing additional code, called *serializers*, that dynamically determine the data on which a computation operates. The runtime system (in the form of library support or a managed runtime environment) executes the serializer before a computation occurs, mapping operations on the same data to the same serialization set, and computations on different data to different serialization sets. Members of the same serialization set are executed sequentially to honor data dependences, and members of different serialization sets may be executed concurrently to exploit dynamic independence. As a consequence of the serial ordering of operations on each data structure, parallel execution of programs written with serialization sets is deterministic, without requiring the programmer to use locking. Writing serializers places a modest additional burden on the programmer, but is significantly easier than reasoning about correct synchronization of threads. We believe that serialization sets can provide programmers with a gentle path to unlocking the performance potential of multicore processors, while encouraging the use of modern programming languages and practices.

To study the implementation and performance of programs parallelized with serialization sets, we have written a C++ library called Prometheus (Section 2.2). Using Prometheus, programmers write programs in a familiar imperative, object-oriented language. To achieve good parallel performance, Prometheus programs should strive to perform computations on private object state, and structure programs hierarchically. In our experience, this dovetails nicely with the principles of encapsulation and modularity inherent to object-oriented programming.

Prometheus uses C++ templates to instantiate run-time support structures at compile time, and provides a run-time library that can be adapted to the parameters of the execution environment, such as the number and performance of cores (Section 4). Prometheus also provides a library of useful programming tools, including pre-written serializers, and a set of shared data structures. Finally, Prometheus includes support for compiling executables into a debugging version that simulates serialization set execution, so that all development and debugging is done on a sequential program.

We have evaluated the parallel execution of several applications written in C++ using the Prometheus library on a variety of real machines (Section 5), including multi-core, symmetric multiprocessors (SMPs), and cache-coherent non-uniform memory access (ccNUMA) machines running the SPARC-V9 and x86-64 ISAs. We study the scalability and performance limitations of Prometheus programs, and compare our results with existing parallel programming models.

2. SERIALIZATION SETS

To use serialization sets, the programmer divides program execution into two types of phases: *aggregation epochs*, and *isolation epochs*. Aggregation epochs comprise traditional sequential execution, and are the default mode. During isolation epochs, computational operations execute in one of two abstract *contexts*—the *program context*, and the *delegate context*. The program context executes code according to standard sequential semantics. The programmer (or compiler) identifies *potentially* independent computational operations, and these may be assigned (or *delegated*) to the delegate context, which executes them on behalf of the program context.

In an isolation epoch, the programmer partitions data into a number of disjoint domains: *read-only* data that may be freely accessed by any operation; and *privately-writable* data that may be read and written only by its owner. This partition is fixed during a particular isolation epoch, but may be different in subsequent isolation epochs. The program context initially owns all writable data. When the programmer identifies a computational operation for delegation, she associates it with a *serializer*, code that executes at runtime to identify a *serialization set* for the operation. The serialization set becomes the owner of the writable data accessed by the operation until the program context reclaims ownership, or the isolation epoch ends.

The programmer must specify the serializer such that all computational operations on the same writable domain are mapped to the same serialization set. She should also endeavor to have the serializer map computations on different writable data to different sets. The delegate context executes operations composing a particular serialization set in serial order, i.e., the order they are encountered in the program. It may execute members of different serialization sets concurrently in order to improve performance. Executing these operations in parallel does not affect the appearance of sequential

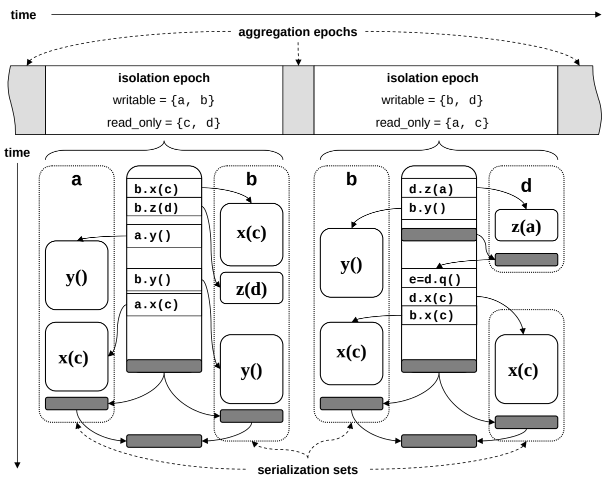


Figure 1: Execution of a program using serialization sets is divided into aggregation and isolation epochs. During an isolation epoch, computational operations (x, y, z) are assigned to a delegate context, where they are applied to data (a, b, c, d). The delegate context applies operations to the data in each set in a serial fashion, and parallelizes operations in different sets.

semantics, because the writable domains owned by various serialization sets are disjoint.

When operations on a particular writable domain are no longer independent, the program context may reclaim ownership of that domain, and then safely intermingle operations on that domain with other domains that it owns. Note that this transfer of ownership need not be specified by the programmer, as it is implicit in the presence of operations dependent on a particular writable domain. At the end of an isolation epoch the program context reclaims ownership of all domains, and may resume traditional sequential execution.

Parallelization using serialization sets results in deterministic execution that is indistinguishable from sequential execution of the same set of operations. Specifically, data races cannot occur because each writable data element is accessed by at most one operation at a time. Other types of concurrency bugs such as deadlock, livelock, and priority inversion, are also precluded, because there is a single logical ordering of all operations, even if the delegate context is overlapping execution of independent computations.

Figure 1 illustrates the execution of a program using serialization sets. In the first isolation epoch, disjoint data elements a and b are designated as writable, and c and d are read-only. Operations x, y, and z are identified as potentially independent and thus may be assigned to the delegate context. For the purposes of this figure, we assume the serializer is specified so that the data elements are all mapped to different serialization sets (e.g. by using the address of the particular element). This enables operations on a to be executed concurrently with the operations on b, as well as with the operations in the program context. All operations are allowed to read (but not write) c and d, as shown in the figure. At the end of the isolation epoch, the program context uses a special operation to reclaim ownership of all domains, as depicted in gray.

The second isolation epoch in Figure 1 uses a different data partition: this time b and d are writable, and a and c are read-only. Operation z on data element d is independent of other operations in the program context, and is thus assigned to the delegate context. Later the program context needs to read part of data element d using operation q. Because q has not been designated as independent, the program context must first reclaim ownership of d as shown before performing this operation. Later, when independent operation `x` is reached, ownership is once again assigned to the delegate context to re-enable concurrent execution of operations on this domain.

2.1 Specifying Serializers

Serializers provide a flexible mechanism for expressing dependences between operations. The programmer or compiler needs no *a priori* knowledge of the dependences between operations that are delegated; the serializer dynamically determines the degree of independence in the operations, which may vary depending on the input to the program. The serializer reveals the degree of independence present in the operations as the program executes, and the Prometheus runtime automatically parallelizes them.

The simplest serializer would identify the data manipulated by an operation, for example by computing the address of the data in memory. For object-oriented programs, the serializer might use a sequence number that reflects the order in which an object was instantiated from a class specification. In some cases, identifying the data accessed by the operation may represent a significant fraction of the work done by the operation, and it is beneficial to use some other mapping of operations to serialization sets, so long as computations on the same data are mapped to the same set.

While it is generally desirable to specify a serializer so that disjoint data elements are mapped to different serialization sets, it may be advantageous to map operations on different elements to the same serialization set. Assigning data elements stored in the same cache line to the same set ensures that they will be operated on by the same processor, alleviating false sharing effects. It may also be useful to assign data elements stored in contiguous memory to the same serialization set to leverage the prefetching mechanisms present in modern processors.

Serializers may be specified as a special operation that is implicitly associated with a data type (an *internal serializer*) and automatically used whenever operations on that type are performed, or they may be explicitly specified at the point of delegation (an *external serializer*). Internal serializers are typically used when the data structure contains identifying information. For example, an object may have a special method that serves as the serializer, computing the serialization set using its internal state. External serializers are used when the identifying information is not stored in the object.

2.2 Effecting Shared State

A key aspect of any parallel execution model is how it manages shared data that must be accessed by multiple processors (aside from the trivial case of read-only data). Because the use of serialization sets requires partitioning data into disjoint domains, it might seem to overly restrict how sharing can occur. Indeed, limiting how data is shared among processors is central to how serialization sets avoid traditional concurrency bugs. The remainder of this section describes three techniques for effecting computation that would require shared state using more conventional parallel models.

The first technique is to use different partitions of data in different isolation epochs, as shown in Figure 1. By alternating which data is in read-only vs. writable domains in an iterative fashion, serialization sets may achieve the effect of coarse-grain sharing.

The second technique leverages the fact that many operations amenable to parallel execution are both associative and commutative, and thus may be performed in any order. We refer to these as *reducible*, because operations may access a local version of the data[1], and a reduce (also known as a fold) operation is performed to summarize these versions into the final result at the end of the isolation epoch. The reduce operation on N elements is performed using $N_{i-1}/2$ parallel operations at each step i. Reducible operations are used in many parallel execution models; notable examples in the realm of imperative programming are Google's MapReduce [3], Cilk's inlets [4], and hyperobjects in Cilk++ [12]. We note that many non-reducible operations may be transformed into reducible operations by deferring the components of the operations that do not commute or associate into the reduction itself.

[1]Because the local version is writable only by single processor, reducible data is thus a special case of privately-writable data.

Third, we note that many operations on shared data are not allowed to execute concurrently under parallel models such as multithreading. Critical sections are used to ensure mutual exclusion, preventing simultaneous uncoordinated accesses that could corrupt the shared data. In this case, the data is never simultaneously accessed by multiple threads at all, and thus calling it "shared" is a misnomer. Consider a hash table that is accessed by multiple threads. Typically a lock on the overall hash table is acquired, protecting the metadata and structure of the hash table while the desired data is located. Another lock is acquired on the underlying data, and then the lock on the hash table itself is released. The time spent in the hash table lock must necessarily be short, lest the entire program be serialized through these accesses. Using serialization sets, accesses to container data structures is performed in the program context, and then operations on the underlying data are assigned to the delegate context. Many operations amenable to shared access in multithreading can be handled in this way, since the necessarily brief accesses to the overall structure do not unduly burden the program context. Furthermore, there is no composability problem for these operations, since they are always performed in the program context.

There are likely some sharing patterns that may not lend themselves to efficient implementation using serialization sets. In practice, we have found that a large number of sharing patterns map nicely onto the techniques described in this section.

3. THE PROMETHEUS C++ LIBRARY

This section describes Prometheus, a C++ template library that implements the serialization set execution model. The use of C++ allows programmers to write parallel applications using a familiar imperative, object-oriented language, using existing compilers and libraries, and provides a path for parallelizing existing sequential programs. Templates are the C++ mechanism for generic programming; briefly, the declaration `template <typename A, typename B, ...>` before a class or function indicates a generic specification that can be instantiated by the compiler when it encounters a use of the class or function, replacing `A, B, ...` with the appropriate types. The use of templates affords Prometheus several advantages. First, the compiler automatically synthesizes the necessary code for run-time support based on the types used for classes and methods involved in parallel execution. Second, Prometheus operates above the type system (rather than casting data through `void` pointers), allowing many programming errors to be caught at compile time. Third, templates provide a mechanism to implement the new language features needed for serialization sets via *template metaprogramming* [2, 20], which provides a Turing-complete language for compile-time execution.

3.1 The Prometheus API

Prometheus uses C++ objects to encapsulate data into disjoint domains. Method calls serve as the granularity of operation that may be delegated, and thus potentially executed in parallel. Prometheus leverages the C++ type system to provide some enforcement of the data partitioning requirements, but provides additional support for detecting errors via template metaprogramming.

Function / Method	
Class	***Description***
`initialize ()`	
global	Initializes the Prometheus run-time library.
`terminate ()`	
global	Shuts down the Prometheus run-time.
`sleep ()`	
global	Puts the threads used to implement the delegate context to sleep.
`begin_isolation ()`	
global	Begins a new isolation epoch. Wakes up delegate context processor resources if necessary.
`end_isolation ()`	
global	Synchronizes the program and delegate contexts, starts new aggregation epoch.
`template <typename R, typename T,` ***`paramtypes...`*****`,`** ***`argtypes...`*****`>`** **`R call (R (&T::method) (`*****`paramtypes...`*****`),`** ***`args...`*****`)`**	
`read_only <T>`	Calls method with specified arguments, returns value of type R. During an aggregation epoch, any method may be called. During an isolation epoch, calling non-const methods results in an error.
`reducible <T>`	Calls method with specified arguments on the current context's view of the object, returns value of type R. The first call in an aggregation epoch causes the reduce method to execute, reducing the multiple views of an isolation epoch to the final view.
`writable <T, S>`	Calls method with specified arguments, returns value of type R. Valid use includes calls to *const* methods when object is in a read-only state, or calls to any method when object is in a private state. Other uses generate an error.
`template <typename T,` ***`paramtypes...`*****`,`** ***`argtypes...`*****`>`** **`void delegate (void (&T::method) (`*****`paramtypes...`*****`),`** ***`args...`*****`)`**	
`writable <T, S>`	Assigns a potentially independent method call to the delegate context in the serialization set computed by executing the serializer method of class S. If object is in the read-only state, generates an error. Valid parameter types include native types passed by value, and subtypes of shared passed by reference or pointer. Delegated methods must have a return type of void and arguments must either be passed by value, or must be pointers or references to classes derived from shared.
`template <typename T,` ***`paramtypes...`*****`,`** ***`argtypes...`*****`>`** **`void delegate (ss_t serializer, void (&T::method) (`*****`paramtypes...`*****`),`** ***`args...`*****`)`**	
`writable <T, S>`	Assigns a potentially independent method call to the delegate context in the serialization set specified by the serializer argument. If object is in the read-only state, generates an error. Valid parameter types include native types passed by value, and subtypes of shared passed by reference or pointer. Delegated methods must have a return type of void and arguments must either be passed by value, or must be pointers or references to classes derived from shared.
`template <typename T,` ***`paramtypes...`*****`,`** ***`argtypes...`*****`>`** **`void doall (vector <writable <T, S> > v, void (&T::method) (`*****`paramtypes...`*****`),`** ***`args...`*****`)`**	
`writable <T, S>` (static)	Executes method on all objects in vector objs. The specified method must have a return type of void and arguments must either be passed by value, or must be pointers or references to classes derived from shared.

Table 1: The Prometheus API.

Prometheus provides a set of wrapper classes that implement the different types of data domains. These classes inherit from the class `shared`, and are specialized on type they wrap. The wrapper classes wall off objects and mediate all method calls so that the safety of operations on them can be monitored via a combination of static and dynamic checks. Wrapped objects must be constructed inside the wrapper; they cannot be created by passing in a pointer or reference to an existing object. This prevents the programmer from accidentally using the unwrapped object to perform unchecked calls. The programmer calls methods on wrapped objects using the `call` interface, which accepts a pointer to the desired method in the underlying object and the arguments to the method. After performing the necessary checking, `call` executes the specified method.

During an aggregation epoch, calls to all methods are allowed through any wrapper type. During isolation epochs, the wrapper classes provide special handling for calls according to the type of the wrapper. The `read_only` wrapper allows only calls to `const` methods[2]. The `reducible` wrapper class performs the call on a local view of the object that may differ in different instances of methods on the object; the first call to a method in the following isolation epoch executes the `reduce` method specified by the user to summarize the effects of the parallel operations into the final state. The `writable` wrapper allows an object to be treated as read-only or privately-writable, but not both, for the duration of an isolation epoch. When used as a read-only object, calls to `const` methods are allowed anywhere, but calls to non-`const` methods generate an error. When used as a privately-writable object, independent methods can be assigned to the delegate context via the `delegate` interface; later calls through the `call` interface automatically reclaim ownership of the object before executing. The `writable` class maintains a state machine that signals an error if the object is treated as read-only and privately-writable in the same isolation epoch.

The `writable` class is also specialized on a class that implements the serializer for the object. The programmer may select from a set of predefined serializers provided by the Prometheus library, or they may write their own. The predefined serializers include the *object* serializer, which serializes on the address of an object, the *sequence* serializer, which serializes on the instance number of the object, and the *null* serializer, which is used when

[2] In C++, `const` methods are not allowed to modify the data members of an object.

Embarrassing parallelism

```
vector <writable <object_t> > objects;
writable <object_t>::doall (objects, &object_t::method, args…)
```

Data parallelism

```
vector <writable <object_t> > objects;
for (i = 0; i < objects.size (); i++) {
  objects[i].delegate (&object_t::method, args…);
}
```

Task parallelism

```
writable <object_A_t> object_A (args…);
object_A.delegate (&object_t:start ());
writable <object_B_t> object_B (args…);
object_B.delegate (&object_t:start ());
```

Pipeline parallelism

```
vector <writable <object_t> > objects;
for (i = 0; i < objects.size (); i++) {
  objects[i].delegate (&object_t::pipe_stage_1, args…);
  objects[i].delegate (&object_t::pipe_stage_2, args…);
  objects[i].delegate (&object_t::pipe_stage_3, args…);
}
```

Figure 2: Prometheus implementation of common parallelization schemes.

external serialization is desired.

The key parts of the Prometheus API are shown in Table 1. In addition to the wrapper class methods, the API provides methods to initialize and terminate the Prometheus run-time. For long aggregation epochs, the API provides a `sleep` method that can be used to temporarily release the processor resources used by the delegate context. Isolation epochs are delimited with the `begin_isolation` and `end_isolation` methods.

Prometheus only allows delegation of methods with certain signatures. The first requirement is that the return type must be `void`. Allowing return values would require the program context to wait for delegated methods to complete before continuing execution, which is contrary to our goal of concurrency. To achieve better performance, programmers should restructure independent methods to store return values inside the object, and provide a method to read results at a later time.

The other restriction on methods that can be delegated is that the types of their arguments must meet certain requirements. Any argument may be passed by value, although this is only advisable for primitive types, because copying large structures is expensive. If an argument is passed through a pointer or reference, its base type must be `shared`; in other words, only wrapped objects may be passed by pointer or reference. Furthermore, `read_only` arguments may only be passed by `const` pointer or reference.

The goal of the wrapper classes and the restrictions on arguments to delegated functions is to ensure that delegated method calls cannot interfere with each other, under the assumption that the state of distinct objects does not overlap. This is guaranteed for data inside the object, but if objects contain pointers to outside state they may interfere with each other. Therefore Prometheus also provides a set of smart pointer types that can track ownership of pointed-to objects, and detect errors when they are accessed by more than one owner in an isolation epoch.

Figure 2 gives sketches of several parallelization schemes, as they would be implemented in Prometheus. However, serialization sets do not restrict programmers to these methods, and give flexible support for exposing dynamic independence in many other ways.

3.2 An example Prometheus program

Figure 3 shows a simplified Prometheus implementation of the `reverse_index` benchmark. `reverse_index` recursively reads a directory tree containing HTML files, extracts the links, and produces an index of all files that contain each link. This benchmark is a C++ reimplementation of a program in the Phoenix suite [18].

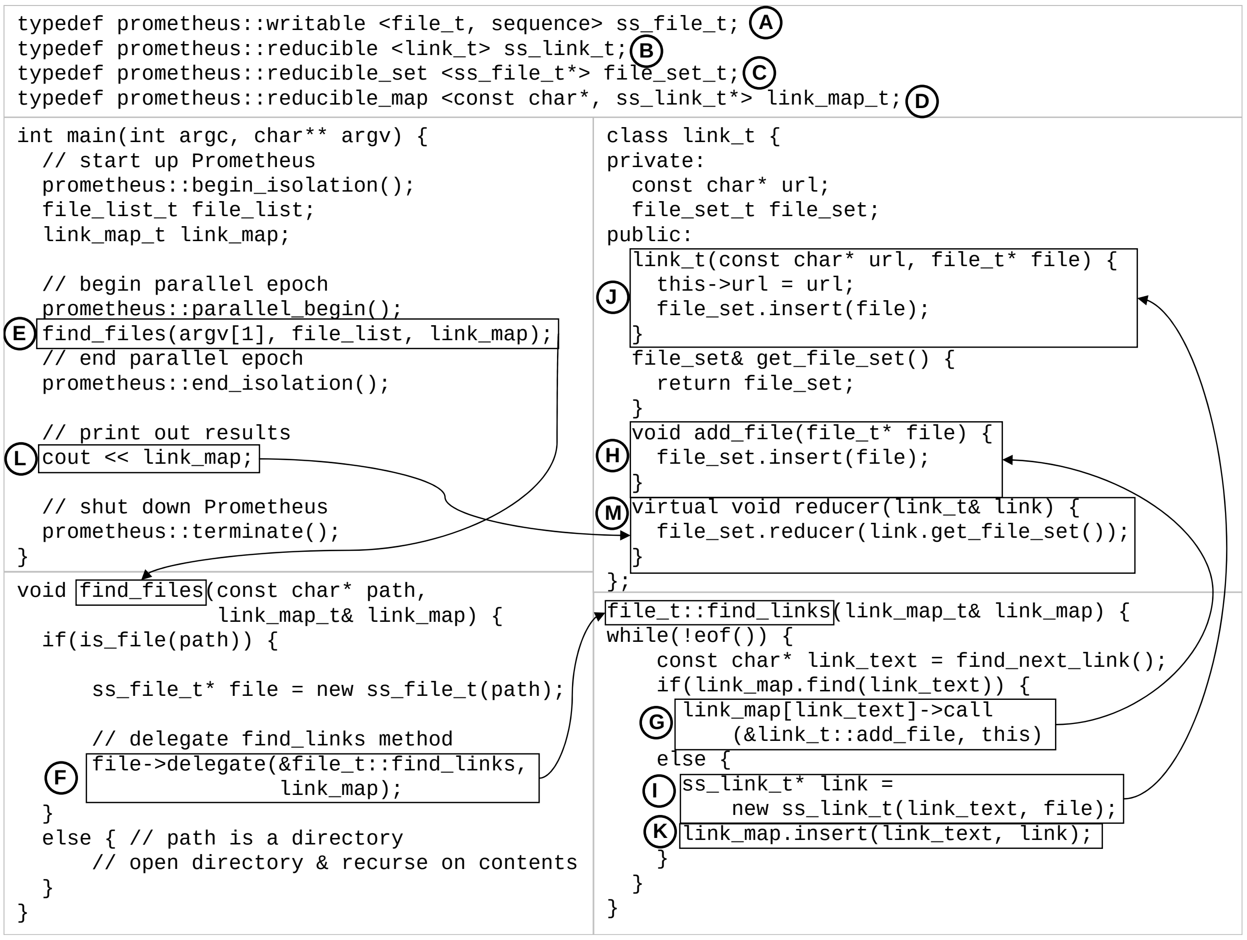
The classes used in this benchmark include a `file_t` class, which stores the path to a file, and implements all the necessary operations on the file, including the `find_links` method. In the Prometheus implementation, the `file_t` class is wrapped in the `writable` class **(A)**, using the `sequence` serializer, which adds an instance number to each object which will be used as the serialization set identifier. Using the `writable` wrapper around `file_t` allows for delegation of method calls. The `link_t` class stores the URL of the link, as well as the set of files in which the link has been found. Operations on objects of this class are reducible **(B)**, because adding files to the set in which the link has been found can be performed in any order. This program also uses two data structures from the Prometheus library: a `reducible_set` **(C)** to store files in the link class, and a `reducible_map` to look up the `link_t` object for a particular text link found in the file.

To parallelize `reverse_index` with serialization sets, our strategy is to start an isolation epoch with `begin_isolation`, and recursively traverse the specified directory using the `find_files` function **(E)**. When a file is found, a new `file_t` object is created, and the `find_links` method of this object is delegated **(F)**. This allows the searching of different files to occur in parallel. Note that with serialization sets, the parallel portion of the program (searching files for links) is overlapped with the sequential part of the program (locating the files). Since the number of files is initially unknown, a typical thread-based implementation would first have to locate all the files, then parcel them into equally-sized sets to evenly distribute work to the threads. This example demonstrates that, while fine-grained parallelization must amortize overheads over smaller units of work, it can extract more concurrency from the program.

The `find_links` method scans through the file searching for links. When it finds a link, it checks the link map to see if the link has been previously encountered; if it has, it adds the current file to the existing link in the link map **(G), (H)**. If the link has not been seen before, a new `link_t` object is created **(I), (J)** and inserted into the link map **(K)**.

Once all the files have been located by `find_files`, the function returns, and then `main` calls the `end_isolation` method. This causes the program context to wait until all outstanding methods have completed in the delegate context, and then reverts to an aggregation epoch. The next step in `main` is to print out the link map **(L)**, and this first use of the link map causes its `reduce` to be called. The reduction finds instances of the same link in different views of the link map, and calls their `reduce` method **(M)** to merge them together. The links are combined by merging their file sets. When the reduction is complete, the link map contains the final index from links to the files that contain them.

The Prometheus implementation of `reverse_index` illustrates



```
typedef prometheus::writable <file_t, sequence> ss_file_t;
typedef prometheus::reducible <link_t> ss_link_t;
typedef prometheus::reducible_set <ss_file_t*> file_set_t;
typedef prometheus::reducible_map <const char*, ss_link_t*> link_map_t;
```

```
int main(int argc, char** argv) {
  // start up Prometheus
  prometheus::begin_isolation();
  file_list_t file_list;
  link_map_t link_map;

  // begin parallel epoch
  prometheus::parallel_begin();
  find_files(argv[1], file_list, link_map);
  // end parallel epoch
  prometheus::end_isolation();

  // print out results
  cout << link_map;

  // shut down Prometheus
  prometheus::terminate();
}
```

```
class link_t {
private:
  const char* url;
  file_set_t file_set;
public:
  link_t(const char* url, file_t* file) {
    this->url = url;
    file_set.insert(file);
  }
  file_set& get_file_set() {
    return file_set;
  }
  void add_file(file_t* file) {
    file_set.insert(file);
  }
  virtual void reducer(link_t& link) {
    file_set.reducer(link.get_file_set());
  }
};
```

```
void find_files(const char* path,
                link_map_t& link_map) {
  if(is_file(path)) {

      ss_file_t* file = new ss_file_t(path);

      // delegate find_links method
      file->delegate(&file_t::find_links,
                     link_map);
  }
  else { // path is a directory
      // open directory & recurse on contents
  }
}
```

```
file_t::find_links(link_map_t& link_map) {
while(!eof()) {
    const char* link_text = find_next_link();
    if(link_map.find(link_text)) {
      link_map[link_text]->call
          (&link_t::add_file, this)
    else {
      ss_link_t* link =
          new ss_link_t(link_text, file);
      link_map.insert(link_text, link);
    }
  }
}
```

Figure 3: Prometheus example program: `reverse_index` (some details have been omitted for clarity).

how writing programs with serialization sets is different from using traditional multithreading techniques. Rather than thinking about threads of control and managing their interaction, serialization sets require the programmer to encapsulate data into classes and structure programs hierarchically. This thought process is consistent with the principles of object-oriented programming. Contrasted with multithreaded programming, we believe this represents a significant reduction in complexity.

3.3 Detecting and Debugging Errors

Because the execution of programs parallelized serialization sets is deterministic, the process of finding and debugging errors is significantly easier than it is in multithreaded programs. Prometheus provides a mechanism to detect errors in the parallel execution, and the capability to perform all development and debugging on a sequential version of the program.

There are two sources of errors in Prometheus programs. The first is the use of an improper serializer that maps the same object to multiple serialization sets. This condition is detected by tagging each object with the serialization set it is mapped to during the first delegation in an isolation epoch. If a later delegation maps it the object to a different serialization set, the runtime will observe the discrepancy and signal an error. Note that these errors are usually avoided by using the serializers that are provided by Prometheus, which have been thoroughly tested on a large number of programs.

The second type of error occurs when an operation violates the partitioning of data, such as performing a write on a read-only object. Many of these errors are caught by the static and dynamic checking that is performed via the wrapper classes. However, C++ limits the safety that can be enforced on the data partitioning. Using the wrapper classes, Prometheus is able to ensure that all variables passed as arguments to a delegated function respect the data partitioning rules. Unfortunately, we have not yet been able to devise a way to ensure that global variables and pointer data members are wrapped in the appropriate classes. If a programmer neglects to use the wrapper classes in these cases, Prometheus is unable to detect incorrect accesses on these kinds of variables. For the small- to medium-size programs used for our evaluation, it is straightforward to ensure the proper use of wrappers via inspection. However, inspection is not a viable solution for large programs. We plan to develop a lint-like tool that will inspect class specifications to identify cases where unwrapped accesses to pointers or global variables are performed in any class that has methods that may be delegated. Since this may be done with a simple-flow insensitive analysis, there is no significant technical obstacle to identifying this kind of error.

All development and debugging of Prometheus programs is done on a sequential execution of the program. Using a compile-time flag, programs may be compiled into a debug version that simulates a serialization set execution by tracking the context and serialization set of each operation. Debugging errors in serializers and reductions is therefore no more difficult than debugging any other type of sequential code. When the debug version executes correctly for a given input, the parallel version will too (with the exception of the current limitations with respect to global variables and pointer members, which are detectable in other ways).

4. IMPLEMENTATION

There are two main components to Prometheus: a set of templates used to instantiate the data structures used for serialization sets, and a runtime that orchestrates the parallel execution using these structures. The runtime currently supports x86 and x86_64 under Linux, and SPARC-V9 under Solaris. The system-specific components are confined to a few files to facilitate porting to other architectures.

The `initialize` function starts up the Prometheus runtime. The initial thread of execution, or *program thread*, implements the program context. The runtime detects the number of processors in the system, and spawns a number of additional *delegate threads* to implement the delegate context. The number of delegate threads is one less than the number of processors by default, but may be configured to some other number via an environment variable. The program thread and delegate threads are bound to distinct processors to ensure performance isolation.

The runtime then initializes a communication queue between the program thread and each delegate thread. The communication queue is based on FastForward [5], a cache-optimized lock-free concurrent queue, which performs very low overhead data transfers between processors. Prometheus augments FastForward with a polymorphic interface to allow multiple types of data to be communicated via the same queue. Because the queues are single-producer (the program thread) and single consumer (a delegate thread), the only synchronization required is checking the full condition on the producer side, and the empty condition on the consumer side. Access to the communication queues is performance-critical, so these conditions are checked in a spin loop rather than using blocking OS synchronization, which would incur prohibitive overheads. The x86 and x86_64 implementations insert the `PAUSE` instruction in these loops to limit consumption of processor resources on multithreaded cores.

The communication queues serve three purposes. First, they transfer the data needed to execute the method in the delegate context. Second, they preserve the ordering of operations in the same serialization set. Third, they provide buffering to help tolerate bursts of operations mapped to the same serialization set.

The presence of a call to `delegate` in the program causes the compiler to instantiate an *invocation class* for the specified method call. The invocation class contains a pointer to the object and method to be delegated, as well as the specified arguments. It also contains the serialization set identifier to allow the runtime to detect erroneous serializers. All invocation classes share a common `execute_method` interface, and the different types of invocations implement this interface to correctly call the method stored in the invocation. Because the invocation classes are instantiated via templates, many type errors (such as calling a method with the wrong arguments types) are detected at compile time. If we had chosen to implement invocation objects with `void` pointers, these errors would not be detected until run-time.

The program thread executes `delegate` directives in four steps. First, it executes the serializer to compute the serialization set number for the method. Second, it performs *delegate assignment*, which identifies the delegate thread that will execute the method. Third, it allocates an invocation object of the appropriate class. Fourth, the invocation object is inserted into the communication queue for the designated delegate thread.

The current Prometheus implementation performs *static delegate assignment*. It takes the modulus of the serialization set number and the number of *virtual delegates*. Because many programs contain small sequential components, the program thread has little work to do compared to the delegate thread, so Prometheus uses the program thread to execute some of the delegated methods. Virtual delegates allow runtime configuration of the *assignment ratio* of serialization sets assigned to the program thread and the delegate thread. The assignment ratio allows for run-time configuration of work distribution to suit the environment the program is running in.

The delegate threads execute a loop to repeatedly read invocation objects from the communication queue. They call `execute_method` on each object, which executes the method. Upon completion of the method, they deallocate the invocation object, continuing on to the next entry in the queue.

Prometheus uses several special kinds of invocation objects to coordinate the execution of the program thread and the delegate threads. *Synchronization objects* are used by the program thread to reclaim ownership of a data domain so that the program thread may perform dependent computations on that object. When the `call` interface of a `writable` object is invoked, it checks to see if there are outstanding delegated method calls on that object. If there are, it executes the object's serializer, identifies the delegate thread operating on the object, sends a synchronization object to that thread, and waits for a response. When the delegate thread reaches the serialization object, it will be the last object in the queue, since the program thread has ceased sending invocations, ensuring that all methods have completed on the object. The delegate thread then invokes `execute_method` on the synchronization object, which signals the program thread that it has regained ownership of the object, and can then execute its call. The same mechanism is used by `end_isolation` to synchronize with all delegate threads, before returning the program thread to an aggregation epoch.

Termination objects are used by the `terminate` function to send messages to all delegate threads. Upon receiving this message, the delegate threads have finished executing outstanding delegate methods; they then signal the program thread that they have completed. Once the program thread receives replies from all delegate threads, it is safe to terminate the program.

Reducible operations are handled by storing the thread id of each delegate thread in thread-local storage. When methods on a reducible object are called, they use the thread id to retrieve the version of the object corresponding to that delegate thread. Later, when the `reduce` method is called, it performs the reduction to summarize the various versions into the final result.

While the current runtime implementation is fairly simple, it yields very good performance on the programs we have studied so far. In the future, we plan to extend the runtime to support hierarchical execution of delegates, and some form of dynamic delegate assignment or work stealing to further increase the kinds of applications that can be efficiently executed with serialization sets.

5. EVALUATION

To evaluate serialization sets, we used benchmarks shown in Table 2. We ported these benchmarks to Prometheus by first rewriting them as idiomatic C++ programs, using object-oriented features and STL data structures. We then augmented them with serialization sets support. This required some further modification of the programs, typically restructuring classes to include additional state, so that methods calls could be rendered independent, and thus suitable for delegation.

The programs were compiled with `gcc-4.3.1 -O3 -march=barcelona` for the AMD Barcelona/Linux platforms and with `gcc-4.2.1 -O3 -mcpu=v9` for the SPARC-V9/Solaris platforms. All programs were compiled as 64-bit executables.

Program	**Source**	**Description**	**Baseline**	**Inputs (S/M/L)**
barnes-hut	Lonestar [9]	N-body simulation	pthreads	(1,000, 25) / (10,000, 50) / (100,000, 75) bodies, steps
blackscholes	PARSEC [1]	Financial analysis	pthreads	(16,384 / 65,536 / 10,000,000) options
dedup	PARSEC [1]	Enterprise storage	pthreads	(31 MB / 185 MB / 673 MB) file
freqmine	PARSEC [1]	Data mining	OpenMP	(250,000 / 500,000 / 990,000) transactions
histogram	Phoenix [18]	Image analysis	pthreads	(100MB / 400MB / 1.4GB) bitmap
kmeans	NU-MineBench [16]	Data mining	OpenMP	(5,000, 50 / 10,000, 100 / 50,000, 100) points, clusters
reverse_index	Phoenix [18]	HTML analysis	pthreads	(100 MB / 500 MB /1.0 GB) directory
word_count	Phoenix [18]	Text processing	pthreads	(10 MB / 50 MB / 100 MB) file

Table 2: Benchmarks used in experimental evaluation.

	x86 Multicore AMD Phenom 9850	**x86 ccNUMA AMD Opteron 8350**	**SPARC Multicore Sun Fire T2000**	**SPARC SMP Sun Fire V880**
Processor Type	AMD Barcelona	AMD Barcelona	UltraSPARC T-1	UltraSPARC-III+
# Processors	1	4	1	8
Cores per Processor	4	4	8	1
Threads per Core	1	1	4	1
Total Execution Contexts	4	16	32	8
Clock Speed	2.5 GHz	2.0 GHz	1.0 GHz	900 MHz
Memory	8 GB	16 GB	16 GB	32 GB
OS	Linux 2.6.18	Linux 2.6.25	Open Solaris	Solaris 9

Table 3: Machine parameters used in experimental evaluation.

Note that we do not run `freqmine` on the SPARC machines due to portability issues in the original benchmark code. The dynamic checks performed by Prometheus were disabled for the performance measurements in this section.

5.1 Experimental Results

Table 3 lists the four machine configurations used for our experiments. We studied two systems based on the AMD Barcelona processor, including a four-core AMD Phenom system and a four-socket four-core AMD Opteron system (16 cores total)[3]. We also measured two systems using SPARC processors, an UltraSPARC T-1 (Niagara) based CMP system composed of eight cores with four threads each, and an UltraSPARC-III based SMP machine. All systems used a 64-bit OS capable of utilizing the full system memory.

There are several sources of overhead in Prometheus programs that are not present in traditional parallel programs. Prometheus introduces additional indirect calls, including internal serializers, the `execute_method` interface of the invocation objects, and the invocation of the method pointer stored in the invocation object. Prometheus programs may also perform more loads and stores, due to the use of the communication queues. Thus we expect absolute performance of Prometheus programs to be slightly lower than conventional parallel programs, although we expect similar scalability.

Figure 4 shows the performance of conventional parallel programs (BL) and Prometheus programs (SS) normalized to the execution time of the *original sequential program,* not our C++ port. The harmonic mean of the speedups is given in the final column. Prometheus programs perform nearly as well as the conventional parallel implementations in most cases, and actually perform better for some benchmarks.

Several benchmarks exhibit performance disparities that are not solely attributable to the additional overheads in Prometheus. The `freqmine` benchmark is written in a very low-level hand-optimized style, and we were unable to match its performance with an object-oriented coding style. Our Prometheus implementation of `kmeans` does not perform as well as the original benchmark due to the use of an inferior algorithm. `kmeans` performs clustering of n-dimensional data points by iteratively finding the nearest cluster point for each data point, and then updating the mean of each cluster point. The original benchmark iterates over the points and updates the cluster points at the same time. The Prometheus implementation iterates over the data points and cluster points separately. We believe we can reduce the performance difference by computing partial sums of the cluster means during clustering, and using a reduction to summarize the results at the end of the computation.

Two of the Prometheus benchmarks perform better than their conventional parallel counterparts. As described in Section 3.2, `reverse_index` achieves better results by overlapping the directory recursion with finding links in the files; in the conventional program, searching the files for links does not start until after the entire directory has been read.

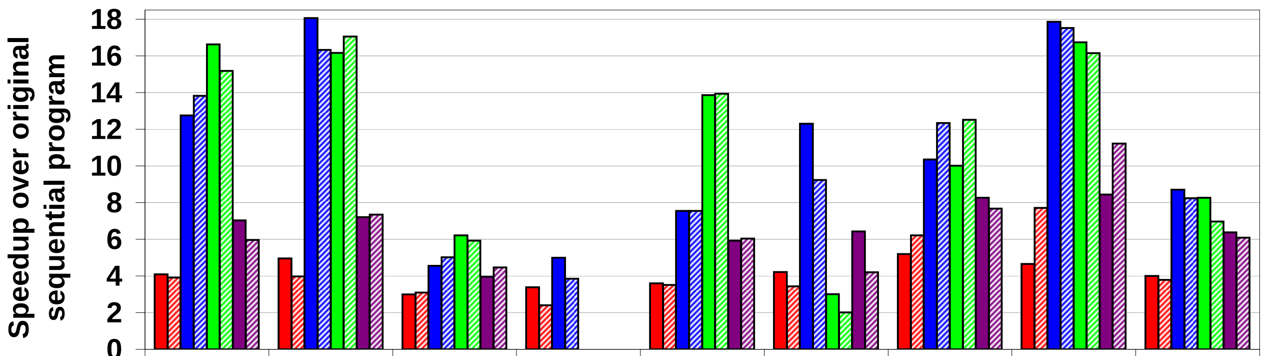
The Prometheus implementation of `word_count` performs significantly better on the machines with four and eight execution contexts, and comparably for the machines with 16 and 32 contexts. The baseline implementation maintains its dictionary of words in a set of lists, and uses all processors in the system to merge different pieces of the lists at the end of the program. The Prometheus implementation uses a reducible map based on the STL data structure, which performs better during the word counting phase, but requires more work to reduce to its final state.

Figure 5a breaks down the amount of time spent in each benchmark into aggregation, isolation, and reduction components, as measured on the 16-core AMD Barcelona system. As expected, better performance of the benchmarks generally correlates to a higher percentage of time spent in isolation epochs, which allow parallel execution. Of the benchmarks that use `reducible` objects, `histogram` spends a negligible amount of time, while `reverse_index` and `word_count` spend about 30% of their execution time in reductions.

Figure 5b shows how the performance of the Prometheus benchmarks on the 16-core AMD Barcelona system scales as larger inputs are used. The one exception is `dedup`, which performs fingerprint-based compression [1]. The speedups of this program depend on how much compression is needed for a particular file; in this case, the medium input file achieves a significantly higher compression ratio than the small and large files do.

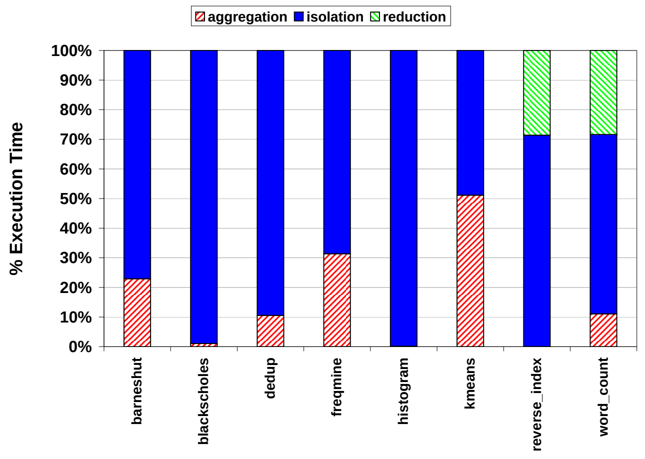
Figure 6 shows the scaling of Prometheus programs on the 16-core AMD Barcelona system. The most parallel program, `blackscholes`, achieves nearly linear scaling. The `barnes-hut` and `word_count` benchmarks both achieve super-linear speedups

[3]We have also run the x86 benchmarks on Intel multicore systems with similar results, but omit the results here for brevity

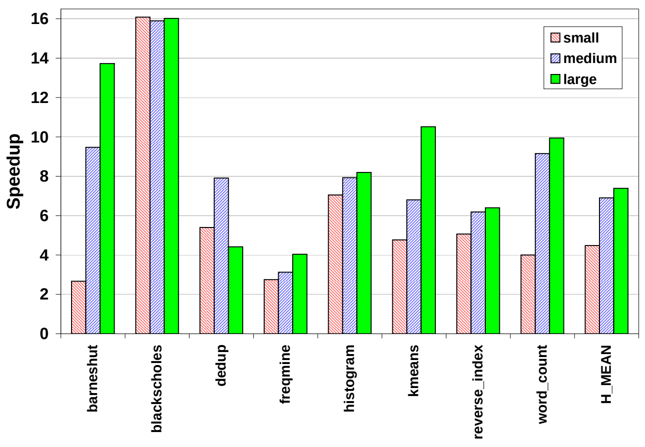


	barneshut	blackscholes	dedup	freqmine	histogram	kmeans	reverse_index	word_count	H_MEAN
■ barcelona-4 BL	4.1	5.0	3.0	3.4	3.6	4.2	5.2	4.6	4.0
▨ barcelona-4 SS	3.9	4.0	3.1	2.4	3.5	3.4	6.2	7.7	3.8
■ barcelona-16 BL	12.8	18.1	4.6	5.0	7.5	12.3	10.4	17.9	8.7
▨ barcelona-16 SS	13.8	16.3	5.0	3.9	7.5	9.2	12.3	17.5	8.2
■ niagara-32 BL	16.6	16.2	6.2		13.9	3.0	10.0	16.7	8.3
▨ niagara-32 SS	15.2	17.1	5.9		13.9	2.0	12.5	16.1	7.0
■ ultrasparc-8 BL	7.0	7.2	4.0		5.9	6.4	8.3	8.4	6.4
▨ ultrasparc-8 SS	6.0	7.3	4.5		6.0	4.2	7.7	11.2	6.1

Figure 4: Performance of conventional parallel programs (BL) versus parallel execution of Prometheus programs (SS).



(a) Execution time breakdown



(b) Input scaling

Figure 5: Characterization of Prometheus programs on the 16-core AMD Barcelona system.

for smaller numbers of threads, but taper off at higher thread counts. The least scalable benchmarks, `dedup` and `freqmine`, do not see additional performance improvement beyond roughly eight delegate threads. Since the conventional parallel version of these programs show the same scaling limitations, we believe the scaling limitations of these programs is due to the algorithm used, and not an inherent limitation of serialization sets.

The scaling of `histogram` is particularly interesting. This benchmark improves in performance up to 10 delegate threads, but then rapidly falls off as more delegate threads are added. We observed similar behavior in the original benchmark as well. (For this reason, the numbers reported earlier for this benchmark use the 10-thread configuration on this system.) Because this program rapidly reads through large portions of memory, we hypothesize that memory bandwidth becomes saturated beyond 10 delegate threads, causing the performance to degrade.

6. RELATED WORK

Like serialization sets, actors [7] and active objects [10] avoid data races by performing operations on data in a single thread. Unlike serialization sets, these models tie objects to a single thread of control, losing the flexibility to be used in different ways, and communicate via asynchronous message passing, resulting in non-deterministic execution.

Halstead's MultiLisp [6] introduced the notion of *futures*, which execute an expression concurrently with the program. This process is similar to delegating a method call, but since futures provide no coordination on shared state, MultiLisp cannot safely futurize multiple expressions that involve the same data.

The Jade [19] language statically divides a program into tasks, and uses *access specifications* for the inputs and output variables of a task to determine when a task is ready to execute. Access specifications are similar to serializers in that they execute dynamically. However, access specifications determine when data is ready for a computational operation, while serializers are used to identify the owner of data to which the operation should be sent. Access specifications must be written for every input and output variable of a task, while only a single serializer is needed to delegate a method call.

Cilk [4], Cilk++ [12], TBB [8], and OpenMP [15] provide a

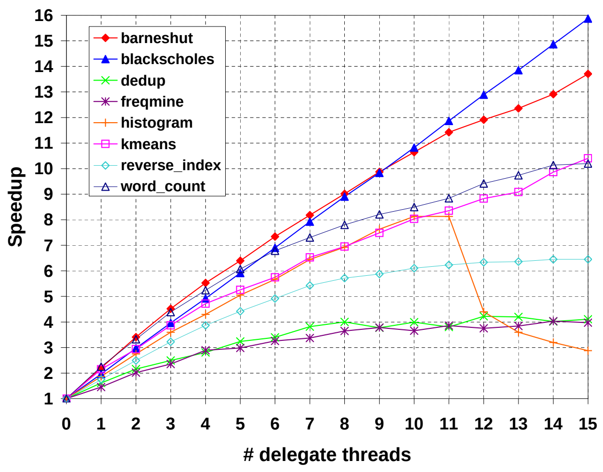


Figure 6: Performance scaling of Prometheus programs on the 16-core AMD Barcelona system.

sequential programming interface for writing multithreaded programs, but like threads, assume that tasks are independent. Explicit synchronization is required to avoid data races on shared state.

Microsoft's Task Parallel Library [14] and the .NET ThreadPool [13] provide mechanisms to delegate method calls to other threads, but require manual synchronization of accesses to shared memory.

Google's MapReduce [3] exploits data parallelism by alternately executing a function on each data element that maps it to a key value, and then performing a reduction on all data with the same key. MapReduce sacrifices generality for scalability, and many kinds of applications are not amenable to the strict map-reduce data flow.

Transactional Memory (TM) [17] provides atomic execution of critical sections by dynamically detecting and undoing the results of conflicting accesses. TM improves the ease of multithreaded programming by providing composability and extracts the performance of fine-grained synchronization from coarse-grained synchronization. TM is not a panacea, and still requires programmers to correctly identify critical sections and reason about nondeterminism.

7. CONCLUSION

Serialization sets offer a new approach to achieving parallel execution of programs, allowing programmers to unlock the power of multicore processors. Instead of requiring the programmer to reason about the nondeterministic execution of multiple threads of control, they encourage hierarchical program structure, and encapsulation of state within objects. Programmers expose independence by providing succinct serializers which dynamically classify operations into dependent sets, and allow a dynamic runtime to schedule independent operations in parallel.

We have described Prometheus, our initial implementation of the serialization sets model as a C++ template library. The results are promising—Prometheus programs perform as well or nearly as well as threaded implementations, with significantly lower programming complexity. In the future, we plan to improve Prometheus to offer more features to exploit dynamic independence, as well as to study more complex programs to see how they mesh with the serialization sets paradigm.

Acknowledgments

This work is supported in part by National Science Foundation (NSF) grants CCF-0702313 and CNS-0551401, a donation from Intel Corp., funds from the John P. Morgridge Chair in Computer Sciences and the University of Wisconsin Graduate School. Sohi has a significant financial interest in Sun Microsystems. The views expressed herein are not necessarily those of the NSF, Sun Microsystems or the University of Wisconsin.

8. REFERENCES

[1] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, 2008.

[2] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Addison-Wesley, 2000.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. of 6th OSDI*, 2004.

[4] M. Frigo et al. The implementation of the Cilk-5 multithreaded language. In *Proc. of PLDI*, pages 212–223, 1998.

[5] J. Giacomoni et al. FastForward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue. In *Proc. of 13th PPoPP*, pages 43–52, 2008.

[6] R. H. Halstead. MULTILISP: A language for concurrent symbolic computation. *ACM TOPLAS*, 7(4):501–538, 1985.

[7] C. Hewitt. Viewing control structures as patterns of passing messages. *Journal of Aritificial Intelligence*, 8:323–363, 1977.

[8] Intel. Threading building blocks. `http://threadingbuildingblocks.org`.

[9] M. Kulkarni et al. Optimistic parallelism requires abstractions. In *Proc. of PLDI*, pages 211–222, 2007.

[10] R. G. Lavender and D. C. Schmidt. Active object: An object behavioral pattern for concurrent programming. In *Proc. of 2nd PLoP*, 1995.

[11] E. A. Lee. The problem with threads. *IEEE Computer*, 39(5):33–42, May 2006.

[12] C. E. Leiserson. Cilk++: Multicore-enabling legacy C++ code. Carnegie Mellon University Parallel Thinking Series, April 2008.

[13] Microsoft. Programming the ThreadPool in .NET. `http://msdn.microsoft.com/en-us/library/ms973903.aspx`.

[14] Microsoft. Task parallel library (TPL). `http://msdn.microsoft.com/en-us/magazine/cc163340.aspx`.

[15] OpenMP. The OpenMP API specification for parallel programming. `http://openmp.org/wp/`.

[16] J. Pisharath et al. NU-MineBench 2.0. Technical Report CUCIS-2005-08-01, Northwestern University, 2005.

[17] R. Rajwar and J. Larus. *Transactional Memory*. Morgan Claypool, October 2006.

[18] C. Ranger et al. Evaluating MapReduce for multi-core and multiprocessor systems, 2007.

[19] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM TOPLAS*, 20(3):483–545, 1998.

[20] T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995.