Sprint: Speculative Prefetching of Remote Data

Arun Raman
Princeton University, USA
rarun@princeton.edu

Greta Yorsh
ARM, UK *
greta.yorsh@arm.com

Martin Vechev
ETH Zurich and IBM Research
martin.vechev@gmail.com

Eran Yahav
Technion, Israel *
yahave@cs.technion.ac.il

Abstract

Remote data access latency is a significant performance bottleneck in many modern programs that use remote databases and web services. We present Sprint—a run-time system for optimizing such programs by prefetching and caching data from remote sources in parallel to the execution of the original program. Sprint separates the concerns of exposing potentially-independent data accesses from the mechanism for executing them efficiently in parallel or in a batch. In contrast to prior work, Sprint can efficiently prefetch data in the presence of irregular or input-dependent access patterns, while preserving the semantics of the original program.

We used Sprint to automatically improve the performance of several real-world Java programs that access remote databases (MySQL, DB2) and web services (Facebook, IBM's Yellow Pages). Sprint achieves speedups ranging 2.4× to 15.8× over sequential execution, which are comparable to those achieved by manually modifying the program for asynchronous and batch execution of data accesses. Sprint provides a simple interface that allows a programmer to plug in support for additional data sources without modifying the client program.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Run-time environments

General Terms Design, Languages, Performance

Keywords remote data, automatic, prefetching, parallelization, speculation, batching, caching, compiler, run-time, tool

* This work was done while the author was at IBM Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.
Copyright © 2011 ACM 978-1-4503-0940-0/11/10. . . $10.00

1. Introduction

Web, business, and scientific programs have increasingly become data bound. They issue large numbers of long latency data access requests—long latency because the data is often served by remote web services or databases. Owing to the disparity between CPU speeds and network latencies and bandwidths, these programs spend a significant fraction of their execution time waiting for the data access requests to be serviced.

To improve the performance of such programs, programmers expend a lot of time and effort scheduling the requests in a way that minimizes the overall execution time using schemes such as asynchronous execution, batching, and parallelization. It usually requires significant code rewriting, thereby obscuring the functional logic of the program, and often results in non-portable performance gains. Ideally, the programmer should only be concerned with expressing the functional logic of the program, and allow the compiler and run-time to orchestrate the remote data requests efficiently.

1.1 Addressing Latency Issues via Prefetching

A conventional way to overcome the problem of data access latency is data prefetching [23, 31]. The idea is to issue asynchronous data requests before the data is really needed so that the data may be available locally when accessed by the program. The desired characteristics of a prefetching mechanism are (1) accuracy: make a good prediction about what remote data will be accessed by the program, (2) effectiveness: make the remote data available locally by the time the program needs it, and (3) correctness: guarantee that prefetching does not affect the program's behavior. Prefetching has been widely studied in the microarchitecture community to hide the latency between the processing core and the memory subsystem [6, 16, 21, 35]. Prefetching has also been used to hide the latency of a local filesystem [4, 18]. The ratio of latency of data access and latency of computation is relatively low in these domains. Consequently, prefetchers in these domains need run only slightly ahead of the main thread of

computation, and overlapping just a single data access with computation often suffices. By contrast, prefetchers for remote data must be able to overlap several remote data accesses since the ratio of latency of data access and latency of computation is very high.

Most prefetchers are history-based: they analyze data access patterns performed in the past, predict that future data accesses will follow similar patterns, and prefetch the corresponding data [8, 10, 17, 18, 25]. While this approach works for programs with regular data access patterns, such as array-based scientific programs, it is not effective for programs whose data accesses depend on the input, are not structured in easily predicted patterns, or do not contain recurrences (that is, frequent reuse of the same remote data).

In a departure from history-based prefetching, Chang and Gibson use speculative execution to allow programs to dynamically discover future read accesses to disk [4]. In the presence of dependencies between accesses, their approach often causes misspeculation of future disk accesses, and spurious disk accesses. Speculative parallelization schemes [9, 19, 33] provide a mechanism for detecting violations of control and data dependencies and use re-execution to guarantee that all dependencies are respected. These schemes offer hope for issuing multiple remote data requests in parallel, but have some disadvantages in our setting. In particular, violation of dependencies that do not contribute to generating remote data requests may cause re-execution, thus re-executing some expensive remote access and hindering progress towards exposing other requests for remote data. It is therefore important to distinguish between dependencies that matter for identifying remote data accesses, and those that matter for computation that uses the data returned by the accesses.

1.2 Prefetching from Remote Data Sources

We propose an approach, implemented in a tool called Sprint, that separates the concerns of exposing potentially-independent remote data accesses from the mechanism by which the overall completion time of the data accesses is reduced. To expose remote data requests, our approach relies on a prefetcher, automatically constructed from the original program. To schedule these requests efficiently, our approach relies on an execution engine (see Figure 1).

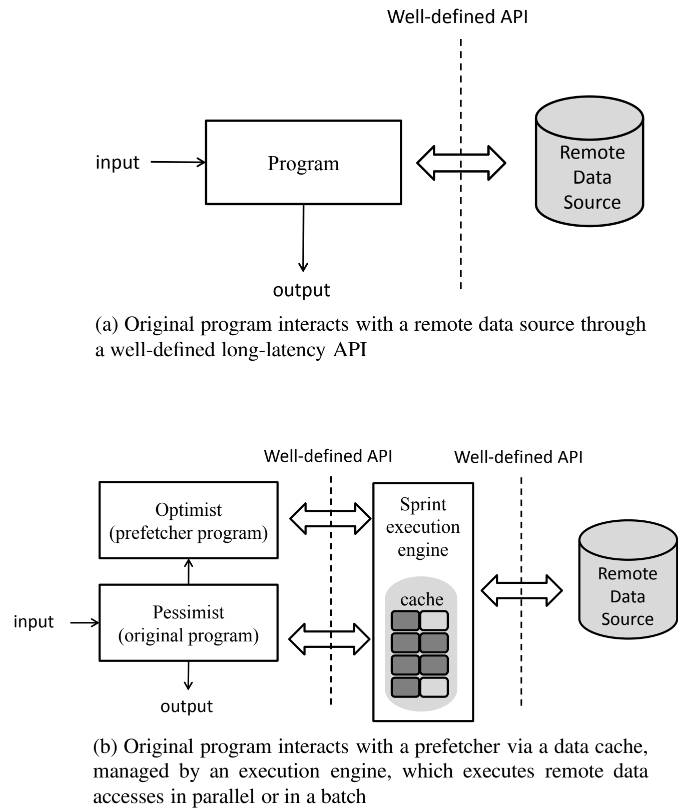

Figure 1. Conceptual model of execution (a) without Sprint (b) with Sprint

Prefetcher The prefetcher is essentially a copy of the original program, that executes concurrently with the original. The prefetcher and the original program issue remote data requests and cache the returned data locally in a shared data-structure. The prefetcher executes faster than the original program, because it is executed by multiple threads in a speculatively-parallel manner, whenever resources are available. In the common case, the original program finds that the data it requires is already available in the cache.

By restricting the effects of the prefetcher to the cache, the behavior of the original program is preserved. By constructing the prefetcher from the original program, all dependencies are respected unless an explicit decision to speculate is made. The prefetcher is designed to respect dependencies between remote data accesses, but may violate other dependencies to expose potentially-independent data accesses. This approach works well even for programs with irregular and input-dependent data accesses.

Execution engine The execution engine manages the cache and automatically decides when to batch up accesses and when to issue them in parallel. Additionally, the execution engine is responsible for monitoring the effectiveness of the speculative prefetcher and restarting the prefetcher if the speculation goes astray (because some remote data accesses were incorrectly dubbed as independent). Fortunately, monitoring cache misses in the execution of the original program provides a low-overhead way to identify violations of dependencies between remote data accesses.

If the remote data source may be modified by other entities (not the program executed under Sprint), then the execution engine keeps the cache in sync with the data source using the mechanism of triggers and callbacks. If this mechanism is not supported by the API of the remote data source (databases usually support it, but remote services may not), then the programmer is responsible for guaranteeing that the

program does not rely on any data consistency constraints to be maintained by the remote data source.

Interaction with the programmer Our goal is to reduce the burden on the programmer by helping to avoid code rewriting that obstructs the functional logic of the program and results in non-portable performance gains. Towards this end, we designed a practical system that can automatically perform speculative prefetching and optimization of remote data accesses, implemented using certain standard APIs, such as JDBC and HTTP. Our system uses profiling (on representative inputs supplied by the programmer) and run-time monitoring methods (Section 5) to automatically identify remote data accesses that are likely to be independent, and to decide when to start and stop speculative prefetching.

1.3 Contributions

The main contribution of this paper is the design, implementation, and evaluation of an integrated tool called Sprint. Sprint is the first automatic tool to reduce the latency of programs that perform multiple accesses to data from sources such as remote web services and databases, while preserving the semantics of such programs. Sprint consists of the following components:

- **Profiler** that identifies potentially-independent remote data accesses to improve accuracy of speculation
- **Bytecode transformer** that uses the profiling information to automatically generate a program-specific prefetcher that preserves the behavior of the original program
- **Execution engine** that optimizes remote data accesses by executing them in parallel and in a batch, monitors the effectiveness of the prefetcher, and maintains correctness by keeping the local cache in sync with the remote data source
- **Plug-in support for new APIs** that allows programmers to apply Sprint to new data sources without modifying the client programs

We used Sprint to automatically improve the performance of several Java programs that access remote databases (MySQL, DB2) and web services (Facebook, IBM's Yellow Pages), and achieved speedups ranging from 2.4× to 15.8×.

Our approach is not a silver bullet for all latency issues with remote data sources. It is designed for programs that can be characterized by mostly read-only accesses, irregular or input-dependent data access patterns, and very low computation latency to data access latency ratios. Our experimental evaluation in Section 6 shows that for programs in the target domain, our approach automatically provides performance that is comparable to the performance achieved by manually modifying the program for asynchronous and batch execution of remote data accesses.

2. Motivating Example

Consider the example code shown in Listing 1 (ignoring @Launch and @Speculate annotations on lines 16 and 26 for now). When executed, this program displays the management hierarchy rooted at the employee whose email address is the program input. In line 27, the method `buildTree` is called to fetch the corresponding hierarchy subtree from the remote data source into a local data-structure of `Node` objects.

```
1  class Node {
2    static int numNodes = 0;
3    Node tree;
4
5    Node buildTree (String email) {
6      Employee emp;
7      try {
8        emp = getEmployee(email); // Expensive remote data access
9      } catch (EmployeeNotInDatabaseException e) {
10       System.err.println('Employee ' + email + ' not found!');
11       return null;
12     }
13     Node root = new Node(emp);
14     numNodes++;
15     for (String reportee_email : root.getReporteesEmail()) {
16       Node child = @Speculate buildTree(reportee_email);
17       if (child != null) {
18         root.add(child);
19         child.setParent(root);
20       }
21     }
22     return root;
23   }
24
25   void main(String[] args) {
26     @Launch Optimist(buildTree);
27     tree = new Node().buildTree(args[1]);
28     display(tree);
29   }
30 }
```

Listing 1. Example of building a managerial tree

The program uses a high-level API to access the remote data source. From the viewpoint of the program, the remote data source is just a mapping from keys to values. A remote data access is a lookup of the value stored in the remote data source for a given key. In this example, a key is the email address of an employee, and a value is the record of that employee, including the list of email addresses of employees who directly report to the employee.

In `buildTree`, the call to `getEmployee` in line 8 is expensive, because this method accesses the remote data source to fetch the details of the employee (implementation not shown). If the employee is not found, `buildTree` prints an error message and returns. Otherwise, `buildTree` constructs a node that will be the root of the subtree that corresponds to the employee (line 13), increments the count of nodes in the tree (line 14), and iterates over the direct reportees of the employee (line 15). Every iteration recursively builds the subtree that corresponds to the reportee (line 16) and updates the tree by linking the employee and the reportee nodes (line 18 and line 19).

Figure 2 shows the hierarchy computed by a sample execution of this program. In this execution, the program per-

formed a sequence of 9 remote data accesses, which corresponds to the depth-first traversal of the tree in Figure 2.

The total execution time of this program is dominated by the latency of the remote data accesses. Our goal is to reduce the total execution time by overlapping the round trip times of remote data accesses whenever possible, without requiring the programmer to modify code. Existing APIs support parallel and batch access to remote data sources and thus provide a way to overlap round trips. The challenge is to identify as early as possible which remote data is accessed by the program, in the presence of dependencies among remote data accesses.

We say that a remote data access $t2$ depends on remote data access $t1$ if the key used by $t2$ is computed from the value returned by $t1$. For example, the remote access `getEmployee(jacob)` depends on `getEmployee(david)`, because the key `jacob` is computed using the value returned by `getEmployee(david)`. It is easy to see that the dependencies in this example are structured as a tree that mimics the hierarchy shown in Figure 2. The longest chain of dependent remote accesses is of length 3, indicating potential for improvement upon the sequential execution.

Note that there are no dependencies between remote data accesses in different iterations of the loop in line 15, but there are memory dependencies between the loop iterations due to updates of `numNodes` in line 14 and the `Node` datastructure in line 18. In other words, there are two kinds of dependencies—those that are required to determine the key for the next remote data access, and those that are not.

Existing methods are ineffective in this setting because they do not distinguish between these two kinds of dependencies. For example, parallelization of the loop in line 15 using Safe Futures [33] would end up executing all remote data accesses sequentially, because Safe Futures respect all dependencies, including memory dependencies in lines 14 and 18. The speculative execution method proposed by Chang and Gibson [4] would speculate the return value of the first call to `getEmployee` in line 8, leading to misspeculation of the subsequent remote data accesses that depend on this value. Other methods (e.g., transactional memory with abstract locking [24], Galois [19], batching [2, 14, 15]) would require the programmer to modify the code or to specify which dependencies are safe to ignore.

3. Sprint Architecture

In this section, we describe the conceptual architecture of Sprint in a platform-independent manner and highlight the design decisions that matter the most for effective prefetching. Section 5 provides more details about our implementation of Sprint for Java, and shows an effective transformation of the code example in the previous section.

Sprint automatically transforms a program with multiple remote data accesses into a well-performing program that

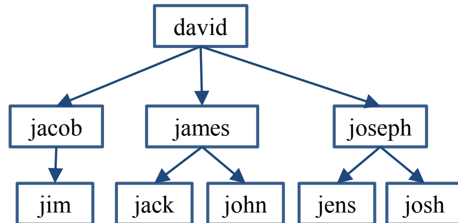

Figure 2. Dependencies between remote data accesses performed during an execution of the example program shown in Listing 1. Assuming sufficient resources, Sprint can reduce the execution latency to the height of the longest dependence chain of remote data accesses multiplied by a round trip latency.

combines the benefits of parallel and batch execution of remote requests. Figure 1 illustrates Sprint's system architecture. Figure 1(a) shows a system where a program interacts with a remote data source through a "well-defined" API (the notion of "well-defined" is explained later), without Sprint.

Figure 1(b) shows the changes with Sprint. The Sprint bytecode transformer makes two versions of the original program, the Optimist (O) and the Pessimist (P). P is nearly identical to the original program, except that at a certain point in its execution, it spawns O and communicates all live-in values for O's execution (indicated by the arrow from Pessimist to Optimist in Figure 1(b)). The idea is for O to serve as a prefetcher for P and to issue remote data accesses as early as possible. Thus, it is important that O runs faster and stays ahead of P. For this, Sprint creates O by speculatively parallelizing one or more loops or recursive methods in the original program. Listing 1 shows annotations at the program points where O is launched (line 26) and O will be speculatively parallelized (line 16). For parallelized execution of O, Sprint's execution engine includes an intelligent thread pool and task queue (Section 3.1).

O and P communicate via a data cache (inside the execution engine) that contains key-value pairs (Section 3.2). The key corresponds to the URL of some remote data, and the value corresponds to the remote data. By virtue of O's runahead execution, P is likely to find that the remote data is available locally. Speculative stores to memory (other than the cache) by O are dynamically privatized at run-time by Sprint, and the data cache maintains consistency with respect to the data source; this ensures the correctness of P's execution (Section 4).

To execute multiple data accesses efficiently, the execution engine uses the logic shown in Figure 4. It dispatches data accesses in parallel, or batches some accesses together in case the remote data source supports batch execution.

In the expected case, O makes data available to P ahead of time through the data cache. The executions of O and P are overlapped in a pipelined fashion causing the overall speedup in program execution time to be fundamentally

limited only by the larger of (i) the length of the longest dependence chain of remote accesses multiplied by a round trip latency and (ii) the time to execute the original code when all required data is available locally. In Figure 2, assuming sufficient resources, the overall execution latency of the program will be reduced from 9 round-trips to 3 round-trips to the remote data source. This is because O would have executed the subtrees rooted at `james` and `joseph` while P is executing the subtree rooted at `jacob`, thereby completely hiding the latency of six out of the nine remote data accesses.

3.1 Optimist Thread Pool and Task Queue

The Optimist is executed by multiple threads in a thread pool (initially containing just one thread) that has the following property: If there is no thread available to execute a task being enqueued, a new thread is created (without exceeding the maximum thread pool size that is specified).

A task is a unit of parallel work that transitively results in a remote data access. For example, `buildTree` on line 16 in Listing 1 is a task. In some applications, if all tasks in the task queue of the thread pool are treated equally, it might happen that O's threads spend most of their time executing data requests that are logically much later in the sequential execution. Consider the task graph shown in Figure 3 which may be generated by a search over tree-structured data. If O's threads execute subtrees rooted at $T4$ and $T6$, P misses in the cache frequently since it is executing logically earlier data requests. And if P terminates early (while O did not because of misspeculation), a large fraction of the prefetches turns out to be useless.

To increase the likelihood of O fetching at least those data items that will be used by P, the task queue is implemented as a priority queue which assigns higher priority to logically earlier tasks. For example, a task created on iteration 1 of a loop is assigned higher priority than a task created on iteration 2. In case tasks spawn more tasks in a nested fashion, a child task inherits the priority of the parent task, with the priorities of siblings being decided in the logical program order. Using the priority task queue, tasks in the subtree rooted at $T1$ will be executed before tasks in other subtrees, allowing $T3$ to be overlapped with $T2$. This results in a reduction in the total time to find the item.

3.2 Data Cache

The data cache is the sole means of communication between O and P. Remote data accesses by *both* O and P are recorded in the data cache. An entry in the cache is a pair of key and $value$. key corresponds to the URL of some remote data. $value$ corresponds to the remote data that is fetched from the URL. $value$ has metadata that indicates its state: absent, issued, or present. Upon a cache lookup, if the state is *absent*, then a remote data access is issued. If the state is *issued*, the caller is blocked until the data is returned by the remote data source. If the state is *present*, data is returned from the cache. Note that both O and P interact with the cache in an identical fashion. Consequently, either can fetch data for the other. In the uncommon case that O falls behind P, the data fetched by P serves to accelerate O.

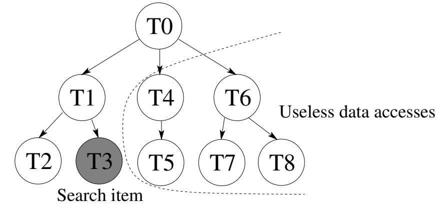

Figure 3. Depending on timing of task (unit of parallel work) execution, several useless data prefetches may be issued in place of useful ones. A priority task queue prioritizes tasks that come earlier in the original program order thereby improving the number of useful prefetches. Tasks are numbered according to their order in the original program.

Batch Execution The execution engine is also responsible for batching remote data accesses. Since there are many threads executing different parts of the program (in both O and P), many entries will be created in the cache for different remote data accesses. Adding another state called *batched* to the metadata of $value$ allows the engine to aggregate all queries in the cache in the *batched* state, issue them all at once, and return the values appropriately thereby releasing the callers that were blocked. This capability of the execution engine frees the programmer from the onerous task of identifying queries to batch and writing code to match the return values of each query with the appropriate point in the original program. To avoid deadlock when the batch limit has not yet been reached and the application will not issue any more queries, a batch flush operation is inserted at the end of the transformed code region. This is described in more detail in Section 5.2.4.

Figure 4 describes how a data request is processed.

Prefetch Throttling O executes the program speculatively and may go down execution paths that differ from the original program's execution, or prefetch remote data that is never used. This may have negative effects such as contention for bandwidth to the remote data source between P, O, and other entities that access the remote data source. To minimize such effects, the execution engine can throttle the speculation by using information available in the data cache as a proxy for the degree of misspeculation. Specifically, if the cache miss rate exceeds a threshold, then the prefetcher is deemed to be unhelpful, and is shut down.

4. Correctness of Sprint Execution

In this section, we define what it means for a program to execute correctly under Sprint, and show how the Sprint architecture described in Section 3 ensures correctness.

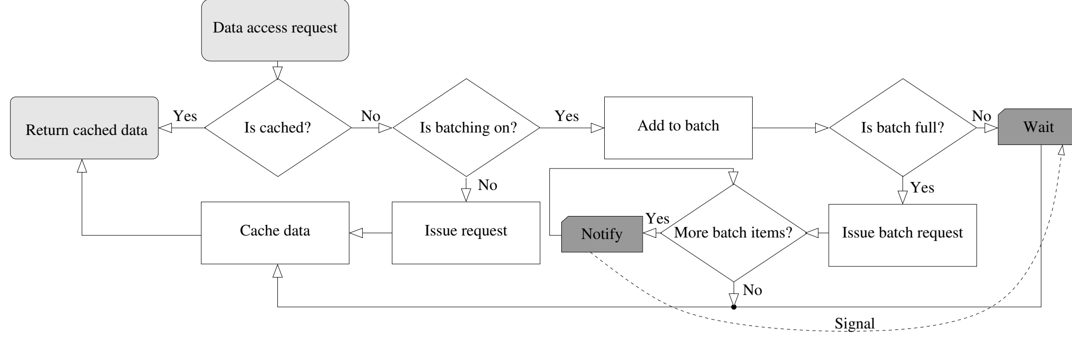

Figure 4. Data request processing algorithm: A data access request is serviced by the cache if data has been prefetched. Otherwise, a data request is issued if batch size equals one, else the request is queued up in a batch and the requesting thread waits. When the batch becomes full, a batch request is issued; when it returns, waiting threads are notified and data is cached. A batch is flushed at the end of a transformed code region.

4.1 Sprint Guarantees

The Sprint bytecode transformer provides the following two guarantees (Section 5.2 describes how):

1. Non-interference through memory: The Optimist and Pessimist execute in the same process, hence the same virtual memory space. The Sprint bytecode transformation ensures that O's actions cannot affect P's memory state: (i) P interacts solely with objects created by itself, and (ii) O only updates the objects that it creates, but may also read objects created by P (indicated by arrow from P to O in Figure 1(b)).
2. Suppression of externally-visible behavior: The transformed program must generate the same sequence of externally-visible behavior as the original program would have generated. The Sprint bytecode transformer ensures this by eliding all side effecting operations from O. In practice, this means catching all exceptions, removing statements such as `System.out.print`, and API operations that may modify the remote data source. To identify such API operations, Sprint requires that the API be "well-defined". For every method whose execution has any observable effects on the remote data, Sprint must know what are the keys corresponding to the modified values.

4.2 Preliminaries

A remote data source is a mapping $M\colon K \to V$ from keys to values. Let R be a (possibly concurrent) program that accesses a remote data source M via a well-defined API. The API of the remote data source supports operations for blocking lookup and update operations that can be described by key-value pairs $r[k, v]$ and $w[k, v]$, respectively.

An execution of R is a sequence of transitions, each of which corresponds to an execution of a basis statement of R or a key-value pair that corresponds to a lookup or update operation on M called by R. A transition $\langle a, b, st, tid\rangle$ consists of source state, target state, statement of R, and thread id. A state $\langle h, s\rangle$ of the program R contains a heap h and a mapping from thread id to the corresponding stack. We assume that M guarantees sequential consistency, and therefore, all the operations on M performed by some execution of R can be ordered. The operations on M performed by an execution t of R form a sequence of key-value pairs $S(t, M) \stackrel{\text{def}}{=} \langle k_1, v_1\rangle, \langle k_2, v_2\rangle, \ldots$. For any sequence S (of pairs or transitions), and for $0 \le i < |S|$, let $S[i]$ denote the i-th value in the sequence S, and $|S|$ the length of S.

Let R' be the result of transforming the original program R using the Sprint bytecode transformer. Let P and O denote the Pessimist and Optimist parts of R', respectively. For an execution t' of R', we use t_P and t_O to denote the projection of t' onto the statements in the P and O parts of R', respectively. Let t'' be a restriction of all states of t_P to $\langle h_p, s_p\rangle$, where h_p contains objects allocated by P (excluding the cache) and s_p contains threads executing in P. We say that t'' is the Pessimist execution of t'.

4.3 Proof of Correctness

THEOREM 4.1 (Correctness of Sprint Execution). *For every execution t' of R', there exists an execution t of R such that $t'' = t$. In other words, a Sprint-ed execution t' of a program R' is correct when it can be partitioned into non-interfering Pessimist and Optimist executions and the Pessimist of t' is equivalent to some execution of the original program R.*

Proof: A transition of t' is an operation on M. t' can be partitioned into t_P and t_O precisely when the following conditions hold:

1. Every state $\langle h, s\rangle$ of t' can be partitioned into (i) objects allocated by each of P and O, $h = h_p \uplus h_o \uplus c$, where c encapsulates an implementation of the abstract data type $cache\colon K \to V$, and (ii) threads executing in each of P and O, $s = s_p \uplus s_o$
2. No transition $\langle a, b, st, tid\rangle$ of t' such that st is in P accesses $\langle h_o, s_o\rangle$ (in state a)

3. All transitions $\langle a, b, st, tid\rangle$ of t' such that st is in O preserve $\langle h_p, s_p\rangle$
4. No transition of t_O performs externally-visible operations (such as an update operation on M, an uncaught exception, etc.)

Guarantee 1 (non-interference throughput memory) provided by the Sprint bytecode transformer ensures that Conditions 1, 2, and 3 hold, while Guarantee 2 (suppression of externally-visible behavior) ensures that Condition 4 holds. Thus, the execution t' can be correctly partitioned into t_P and t_O. Note that this does not impose any constraints on the cache.

t'' is the restriction of t_P to $\langle h_p, s_p\rangle$. By construction, P consists of the same statements as R. From this, and from the correctness of partitioning, we can state that starting from the same original state, for each transition induced by a basis statement (a statement which does not interact with M) of R', there exists a trace t of R in which the same basis statement induces the same transition. It remains to show that this holds for non-basis statements as well. We first define when values in the cache are in sync with the remote data source, assuming there are no updates to the remote data source by entities other than the Sprint-ed program. For a state a, $M(a)$ and $cache(a)$ denote the remote data source and the cache mapping in state a.

DEFINITION 4.2 (Consistency of Cache and Remote Data Source). *Let t' be an execution of R' with Sprint, as before. For all $0 \le i \le |t'|$, if $t'[i]$ is a lookup operation on M and a is the source state of the transition $t'[i]$, then for all $k \in cache(a)$, if the metadata of k in $cache(a)$ is present and the value of k in $cache(a)$ is v, then the value of k in $M(a)$ is v. For all $0 \le i \le |t'|$, if $t'[i]$ is an update operation on M, then the metadata of k in $cache(a)$ is set to absent and $M(a)$ is updated, with both operations happening atomically.*

A non-basis statement that updates M can only appear in t_P since the update operations in O are elided. Furthermore, the statement does not change $\langle h_p, s_p\rangle$. For a statement that looks up M, the consistency of the cache and remote data source ensures that the value returned by the cache is the same as what would have been returned by the remote data source; consequently the statement induces the same transition in both R' and R. Thus, for the transitions induced by the non-basis statements of R', the same transitions are induced by the same statements in trace t of R. $\square$

4.4 Accuracy of Prefetching

Theorem 4.1 implies that for every key-value sequence that may be generated by the Pessimist in some execution t' of the transformed program, there exists an execution t of the original program on the same input that generates the same key-value sequence: $S(t, M) = S(t_P, M)$. Accurate prefetching can now be formally defined.

DEFINITION 4.3 (Accuracy of Prefetching). *Let t' be an execution of R' with Sprint. The speculation is accurate if and only if*

- *for every $0 \le i < |t'|$, if $t'[i]$ is a transition of the Optimist and it performs a lookup operation on M described by the key-value pair $[k, v]$,*
- *then there exists a j, where $0 \le i < j < |t'|$, such that $t'[j]$ is a transition of the Pessimist and it performs a lookup operation on M with k,*
- *and there is no update operation on M with the key k in t' between i and j.*

Note that according to this definition, the prefetch is considered accurate even if the cache was invalidated between i and j by other entities. The sequence observed by an accurate Optimist consists of all the key-value pairs in $S(t_P, M)$, albeit possibly in a different order, because the Optimist is a parallelized version of the original program. However, the Optimist is a *speculatively* parallelized version of the original program. Misspeculation of control and data dependencies could result in a sequence $S(t_O, M)$ that consists of key-value pairs that are different from those in $S(t_P, M)$. The expectation is that misspeculation is relatively uncommon (as we show in the evaluation section) and $S(t_O, M) = S(t_P, M)$.

4.5 Correctness in the Face of Remote Updates

Suppose that a program that uses the remote data source is executing with Sprint, while the data source is concurrently modified by other entities. In this situation, a reordering of reads in O's execution might observe an inconsistent state of the remote data and violate some invariant in P's execution. Suppose that the invariant of P relies on some integrity property of the remote data, and that this integrity property is (atomically) guaranteed by all other entities that may modify the remote data source. Consider an execution in which the remote data is modified by another entity between two out-of-order reads performed by O. It is possible that O observes a state of remote data that does not satisfy the integrity property, and is not observable in any execution of the original program.

EXAMPLE 4.4. Consider a data source M with initial state $M = \{a \mapsto 1, b \mapsto 2\}$. The data integrity that is to be maintained by all entities that interact with the remote data source is $M[b] > M[a]$. This is a typical invariant, for example b could be a summary of elements such as a. The following two programs execute concurrently using M:

```
P1() { x=read(M,a); y=read(M,b); assert (y > x); }
P2() { atomic{ write(M,a,2); write(M,b,3) } }
```

In a concurrent execution of P1 and P2, the set of all possible key-value sequences that can be generated by P1 is:

$$S = \{(\langle a, 1\rangle, \langle b, 2\rangle), (\langle a, 1\rangle, \langle b, 3\rangle), (\langle a, 2\rangle, \langle b, 3\rangle)\}$$

Note that the assertion holds in P1 in all three cases. Suppose that P1 is transformed and executed by Sprint while P2 also

executes concurrently using M. If the reads in the Optimist of P1 execute in parallel and happen to be served out of order by the data source, the following sequence of events may occur at the data source:

```
read(b)                  // by Optimist of P1
write(a,2),write(b,3)    // by P2
read(a)                  // by Optimist of P1
read(a),read(b)          // by Pessimist of P1
```

The following key-value sequence is generated by the Optimist of P1 in this execution: $S' = (\langle a, 2\rangle, \langle b, 2\rangle)$. The data source invariant has been violated! If the remote data source supports "trigger" capabilities, Sprint can solve the consistency problem by installing callbacks in the data source for certain operations that update the data source. □

Enforcing Consistency via Triggers and Callbacks During the execution of a program under Sprint, whenever the Sprint execution engine performs a remote data access with some key, it installs a callback in the remote data source that states "notify me when the value that corresponds to this key is updated". Any write operation will cause the callback to be triggered and the remote data source will notify the Sprint execution engine. Upon receiving notification, the Sprint execution engine will invalidate the appropriate entry in the cache.

EXAMPLE 4.5. In Example 4.4, with the operation `read(b)` performed by the Optimist of P1, the Sprint engine installs a callback on key b. The operation `write(b,3)` performed by P2 triggers the callback on the key b. The execution engine then invalidates the entry for b in the cache. Consequently, `read(b)` by the Pessimist of P1 will miss in the cache, and the request will be reissued. The sequence of key-value pairs observed by the Pessimist of P1 is $([a, 2], [b, 3])$ and the assertion in P1 holds. □

Data sources such as the MySQL database provide triggers with the above semantics that could be leveraged by Sprint. In the absence of trigger APIs, Sprint could ask the programmer whether the consistency semantics arising from the read-read and read-write order relaxation is acceptable. Our current implementation focuses on programs with read-only accesses; the invalidation scheme described above for data sources with trigger APIs will be integrated in future work.

5. Sprint Implementation

In this section, we present implementation details of the Sprint profiler that determines candidate methods to optimize (Section 5.1), the Sprint bytecode transformer that transforms the program at run-time based on the profiling results (Section 5.2), and the Sprint interface that a programmer can implement to use Sprint for optimizing programs that interact with data sources other than those that are currently supported (Section 5.3).

5.1 Profiler

Sprint uses profiling to determine suitable program sites to launch the Optimist (Listing 1, line 26) and the program sites at which to speculate (Listing 1, line 16). Without any modifications, the user executes the program of interest on a representative input with the Sprint profiler turned on. The profiler records the calling contexts leading to remote data access method invocations (such as the JDBC `execute` statement for executing SQL queries). The profiler also maintains loop- (or recursion-) sensitive metadata. Specifically, it records whether a loop (or recursive method) transitively invokes remote data access methods. Such statements within loops (or recursive method callsites) are marked as "candidates". At this point, there are two modes of operation:

- Interactive — the user may prune the candidate set
- Automatic — the profiler directly feeds candidate information to the bytecode rewriter

In the interactive mode, the user is presented with a list of candidates. The user puts the `@Speculate` annotation inside a candidate that is expected to not have dependencies between remote data accesses emanating from it (e.g., Listing 1, line 16). If the user annotates incorrectly—there are dependencies between data accesses at run-time—Sprint ensures correct program execution.

In the fully automatic mode, Sprint can infer the `@Speculate` annotation in one of two ways. Sprint can use a dynamic dataflow tracking tool called Pepe [29] that tracks the flow of data through the remote data access methods to build a remote data access dependency graph. The dependency information is maintained in the context of the candidates. Referring to the candidate loop between lines 15–21 in Listing 1, the profiler records the number of dependencies between remote data accesses that are carried around the loop's back-edge. The frequency of dependencies is used to determine the profitability of transforming the loop. Pepe works for JDBC method invocations only. To transform programs that interact with other data sources, Sprint can transform each and every candidate (independently) and then observe the cache statistics on training runs to determine whether it is worthwhile to retain the transformed candidate. Candidates with high cache hit rates would be transformed while candidates with high miss rates would be ignored.

Pepe is not integrated into Sprint as yet. To obtain the results in Section 6, the Profiler output a list of candidates for each program. The top candidate in each program was marked by the user with the @Speculate annotation; Sprint automatically transformed the programs with that single annotation.

5.2 Bytecode Rewriter

We describe in detail below the code modifications for initiating O in P, constructing O, and preserving the semantics of the original program. The Sprint bytecode rewriter uses the ASM class transformation library [3] to augment the classes

that are loaded at run-time. The bytecode rewriter is written entirely in Java, with no modifications to the underlying virtual machine. The code modifications are also illustrated on the running example in the form of high level Java statements for ease of understanding; in practice, the changes are done to Java bytecode.

5.2.1 Initiating the Optimist

Algorithm 1: Initiating the Optimist

```
Input: Program : original program IR
Input: CandidateSet : set of annotated statements
Output: Program with Optimist initiation
foreach candidate ∈ CandidateSet do
    defMethod ← getDefMethod(Program, candidate)
    callsites ← getCallSites(Program, defMethod)
    foreach callsite ∈ callsites do
        argsCopy ← cloneArgs(callsite)
        initiatePrefetcher(defMethod, argsCopy)
```

P is constructed from the original `Program` by modifying `Program` to initiate O at each callsite of the method containing the candidate loop statement or recursive method invocation (see Algorithm 1). Referring to Listing 1, the immediate predecessor of `buildTree` is `main`. Listing 2 shows the change to `main`. In practice, O is executed by a thread pool (Section 3.1), and initiating O means submitting a task for execution by the thread pool (line 18 in Listing 2). The bytecode rewriter inserts code in P to encapsulate the method and its arguments as a task and submit the task to the thread pool.

```
1  class Node {
2    static int numNodes = 0;
3    Node tree;
4
5    Node buildTree (String email) {
6      Employee emp;
7      ...
8      for (String reportee_email : root.getReporteesEmail()) {
9        Node child = @Speculate buildTree(reportee_email);
10       ...
11     }
12     return root;
13   }
14
15   void main(String[] args) {
16     Node t = new Node();
17     try{
18       OptimistTpool.submit(buildTree, t, args[1]);
19     } catch (Exception e){}
20     tree = t.buildTree(args[1]);
21     display(tree);
22   }
23 }
```

Listing 2. Initiating the Optimist

5.2.2 Constructing the Optimist

O is constructed out of the original program. Sprint performs two code transformations to address the issues of interference of O and P through client program memory, and sequence of side effects that are visible to the external world.

Algorithm 2: Memory Protection

```
Input: ClassSet : Set of classes loaded by program
Output: Pessimist protected from Optimist
foreach class ∈ ClassSet do
    addPrivateBoolean(class, "createdByOptimist")
    foreach method ∈ class do
        copy ← cloneMethod(method)
        entryBlock ← getEntryBlock(method)
        addConditionalRedirectTo(entryBlock, copy)
        foreach inst ∈ copy do
            if inst is store to instance field then
                guard ← createGuard
                        (createdByOptimist, true)
                replace(inst, guarded(inst, guard))
            if inst is store to static field then
                delete(inst)
            if inst is store to array then
                bb ← getBasicBlock(inst)
                callinst ← createCallInst(lookupMap,
                                arrayOwnershipMap)
                addBefore(bb, inst, callinst)
                guard ← createGuard
                        (callinstReturnVal, true)
                replace(inst, guarded(inst, guard))
```

The general approach is to create two versions of every method in a class, one for use by O and the other by P. The design choice of method duplication is motivated by a space-time tradeoff, namely that it allows P to be almost as fast as the original sequential program because P's code remains nearly identical to the original program, at the cost of having multiple copies of each method.

Memory Protection (MP) Transformation All writes to class members are protected by guards in the O version of each method. The details of the MP transformation vary depending on whether the type is an array (see Algorithm 2). Non-array types are discussed first. Listing 3 shows the `Node` class from earlier listings after the MP transformation. A new field `createdByOptimist` (line 4) is added to the class to indicate whether the current instance was allocated by O. This field is set during object allocation (lines 7–8). All writes by O to the instance fields of a class are guarded by ownership checks (for example, the write of `tree` by O in `main_Optimistic` on lines 42–44). O is allowed to write only if the method is invoked on a class instance allocated by O (`createdByOptimist` is true). Writes to static class variables are suppressed in the O versions of methods (line 25). Sprint uses a different strategy for array elements since

the array type cannot be extended to incorporate ownership information. For each array allocated by the original program, Sprint allocates a variable that maintains ownership metadata that is updated at the time of array creation. Sprint maintains a map from array to ownership metadata. This map is used to lookup ownership information when an array is being updated. Sprint uses an optimized multi-level lookup table to reduce the overhead of this operation [12].

```
1  class Node {
2    static int numNodes = 0;
3    Node tree;
4    private boolean createdByOptimist;
5
6    Node() {
7      if (Thread.group.equals(SPRINT_TGRP))
8        createdByOptimist = true;
9    }
10
11   Node buildTree (String email) {
12     if (Thread.group.equals(SPRINT_TGRP))
13       return main_Optimistic(args);
14     ...
15   }
16   Node buildTree_Optimistic (String email) {
17     Employee emp;
18     try {
19       emp = getEmployee(email); // Expensive remote data access
20     } catch (EmployeeNotInDatabaseException e) {
21       System.err.println('Employee ' + email + ' not found!');
22       return null;
23     }
24     Node root = new Node(emp);
25
26     for (String reportee_email : root.getReporteesEmail()) {
27       Node child = buildTree(reportee_email);
28       if (child != null) {
29         root.add(child);
30         child.setParent(root);
31       }
32     }
33     return root;
34   }
35
36   void main(String[] args) {
37     if (Thread.group.equals(SPRINT_TGRP))
38       return main_Optimistic(args);
39     ...
40   }
41   void main_Optimistic(String[] args) {
42     Node temp = new Node().buildTree(args[1]);
43     if (createdByOptimist)
44       tree = temp;
45     display(tree);
46   }
47 }
```

Listing 3. Protecting shared memory

Externally-Visible Side Effect Protection (SEP) Transformation To prevent O from performing operations that result in externally-visible side effects, all such operations (Listing 3, line 21) are elided from the O version of each method. Sprint maintains a database of methods to be elided. Another component of side effect protection is exception trapping. Exceptions may be thrown during the course of O's execution that may not have occurred during the execution of the original program. To ensure that such exceptions do not escape to the user, the prefetch initiation method invocation is wrapped in a `try-catch` block (Listing 2, lines 17–19). In a future implementation, wrapping can be performed at finer granularities in the control flow graph in order to allow O to make useful progress beyond local exceptions.

Input Operation Transformation Reading from files and other input operations are typically "outer loop" activities, whereas Sprint typically optimizes inner loop nests/recursions that use the input data. Presently, all input operations are elided from O similar to the SEP transformation above. Only P performs such operations. In case there is an input operation in the optimized loop, a dependence on which leads O astray, in the current implementation O will be restarted by the execution engine after observing several misses in the cache. If dependencies on input operations are expected to be frequent in the optimized loops/recursions, an alternative implementation could synchronize O and P on input operations. Recent work proposes a system that speculates on user input in the context of web prefetching [22]—this is discussed in Section 7.

5.2.3 Optimizing the Optimist

```
1  class Node {
2    static int numNodes = 0;
3    Node tree;
4    private boolean createdByOptimist;
5
6    Node() {
7      if (Thread.group.equals(SPRINT_TGRP))
8        createdByOptimist = true;
9    }
10
11   Node buildTree (String email) {
12     if (Thread.group.equals(SPRINT_TGRP))
13       return main_Optimistic(args);
14     ...
15   }
16   Node buildTree_Optimistic (String email) {
17     Employee emp;
18     try {
19       emp = getEmployee(email); // Expensive remote data access
20     } catch (EmployeeNotInDatabaseException e) {
21       return null;
22     }
23     Node root = new Node(emp);
24     for (String reportee_email : root.getReporteesEmail()) {
25       Node child = OptimistTpool.submit(
26         new Task(buildTree, reportee_email));
27       if (child != null) {
28         root.add(child);
29         child.setParent(root);
30       }
31     }
32     return root;
33   }
34
35   void main(String[] args) {
36     if (Thread.group.equals(SPRINT_TGRP))
37       return main_Optimistic(args);
38     ...
39   }
40   void main_Optimistic(String[] args) {
41     Node temp = new Node().buildTree(args[1]);
42     if (createdByOptimist)
43       tree = temp;
44     display(tree);
45   }
46 }
```

Listing 4. Optimizing the Optimist (Prefetcher)

For O to execute faster than P, Sprint spawns multiple invocations of a Sprint annotation site optimistically in parallel (Listing 4, lines 25–26). Details of `OptimistTpool` (thread pool) creation are left out with the note that it happens when the Java agent is loaded.

The thread executing the continuation of the spawned future does not block on the future returned by the `submit` call (Listing 4, lines 25–26). Instead, Sprint speculates a return value. In the current implementation, the speculated values are the equivalent of the `null` value for different types. A future implementation could use the results of profiling or memoize values from prior invocations for more advanced speculation.

5.2.4 Avoiding Deadlock Due to Batching

Batch execution and the interaction of the Optimist and the Pessimist via the cache introduce the possibility of deadlock: The execution engine builds a batch of queries as the program issues the queries; however, that batch may remain incomplete once the program issues all the queries that it will ever issue. In the absence of an appropriate mechanism, the program will wait for the queries to return while the execution engine will wait for the program to issue more queries to fill the batch, thus resulting in deadlock. To address this, Sprint inserts a *flush batch* operation at the top of the immediate postdominator of the transformed loop or recursive method in P (the Pessimist). This ensures that the last, potentially incomplete, batch is forced to execute even though the batch may not be full, thus allowing P (and hence the Sprint-ed program) to make progress.

5.3 Extending Sprint

Sprint currently supports data sources that are accessed via the Java Database Connectivity (JDBC) API and the Java URLConnection API. The implementation of the data request processing algorithm in Figure 4 is fully parameterized with respect to the key and value types. This allows arbitrary remote data access APIs to be used with the data request processing algorithm in a straightforward manner using the `Cache` and `Batcher` interfaces (see Listing 5).

To extend Sprint to support other data sources, the programmer must implement the `Batch` interface. This is because different data sources differ in the types of queries that can be batched, the means to prepare a batch, and the means to execute a batch. The `Batch` interface abstracts these details away from the batching logic, allowing the programmer to just supply the data source specific batch preparation and execution code. For example, a `JDBCBatch` implementation of the `execute` method of the `Batch` interface for a DB2 database involves preparing a batch statement via `conn.prepareStatement` and executing the batch via `stmt.executeDB2QueryBatch`. The `Batch` interface is never used by the programmer; it is used internally by the data request processing algorithm. Note that the programmer must implement the `Batch` interface only once for each new data source access API; the implementation can be reused across all programs that use the same API. The `Cache` and `Batcher` interfaces are shown only to illustrate their parameterization; the programmer remains blissfully unaware of their existence.

```
1  public interface Cache <K extends Object, V extends Object> {
2    /**Return the cached value corresponding to the key*/
3    V get(final K key);
4    /**Insert a value to cache corresponding to key*/
5    V put(final K key, final V value);
6    /**Remove cached value corresponding to key*/
7    void remove(final K key);
8    /**Flush the cache*/
9    void clear();
10   /**If cached entry corresponding to key already exists, then
11    * return entry; else cache value*/
12   V putIfAbsent(K key, V value);
13 }
14
15 public interface Batcher<T extends Batchable<R>, R> {
16   /**Add offered element to batch*/
17   R add(T obj);
18 }
19
20 public interface Batch<T extends Batchable<R>, R> {
21   /**Execute batch*/
22   void execute();
23   /**Return true if it was possible to add the offered element to the batch,
         else false*/
24   boolean offer(T obj);
25   /**Return true if batch is full and is the first caller, else
26    * false. Must be invoked only after 'offer' returns false.*/
27   boolean isFull();
28   /**Return true if batch is currently executing*/
29   boolean isExecuting();
30 }
```

Listing 5. Sprint interfaces for extensibility

6. Evaluation

Sprint is implemented in Java, and is evaluated on several client programs and data sources. Table 1 lists the different data sources (with abbreviated names we use in the rest of the paper), the APIs used to access them, and the network (local vs remote) between the client machine and the data source. Table 2 lists the client programs (with abbreviated names we use in the rest of the paper), brief descriptions of each, their core algorithms, the data sources with which they interact, and their input sizes. The programs that interact with YP and DB2 make use of the standard Java URLConnection and JDBC APIs respectively. The Facebook client programs use the RestFB API [1] that itself is a Java client of the Facebook Graph and REST APIs and Java URLConnection. All client programs are written in Java (1.6) running on Linux 2.6.32.

The client machine is equipped with a dual core Intel Core 2 Duo processor running at 2.1 GHz with 32KB (I) and 32KB (D) private L1 caches and a 2MB shared L2 cache, and 4GB of DDR2 800MHz RAM. Reported numbers are averages over five runs. For the client programs and data sources studied, the round trip latency to a data source and back dominated overall latency when compared to the latency of processing requests at the data source.

Data Source (Abbreviation)	API	Supports Batching	Network
IBM's Yellow Pages Web Service (YP)	Java URLConnection	×	LAN
DB2 Database (DB2)	Java Database Connectivity (JDBC)	✓	LAN
Facebook Web Service (FB)	Facebook Graph API	✓	WAN

Table 1. Data sources with which the Sprint-ed programs interact

6.1 Speedup

Figure 5 shows the speedup obtained using Sprint. The baseline for all speedup numbers is the wall-clock execution time of the original unmodified sequential program. The execution time of the optimized versions includes the overhead due to run-time bytecode rewriting. The "Manual Opti." bar represents the speedup obtained by manually rewriting the original sequential program for asynchronous execution (via Java Futures) and batch execution. Sprint achieves the same performance improvements *without needing any code rewrite*. Finally, because the programs are not CPU bound, the speedups obtained (from 2.4× to 15.8×) are not limited by the number of cores (two) on the evaluation platform. Specifically, the number of threads in the Optimist's thread pool grows according to whether a thread is available to execute a task being enqueued. This allows many (> 2) remote data accesses (contained in the tasks) to be concurrently in flight.

Batching Optimization For data sources that support batching, data requests could be accelerated beyond the number of simultaneous connections to the data source, while also enabling the data source's query optimizer to plan a better execution of the queries. Particularly in the case of FC (Facebook), the large reduction in the number of round trips via batching yields a huge benefit. The batch size was arbitrarily set to 100. The interaction of batch execution with parallel execution results in a complex performance model and merits investigation [28].

Prioritized Task Execution In the case of Employee Search (ES), the target loop is responsible for searching a tree and the loop terminates as soon as the item being searched is found. Sprint's control speculation mechanism speculates that the loop will continue executing; consequently the Optimist traverses parts of the search space that are not accessed in the Pessimist's search for the given input. Hence, a significant fraction of prefetching by the Optimist goes to waste (see Table 5). Prioritized task execution prioritizes data requests that come earlier in the original program order and increases the likelihood that useful data is prefetched. Prioritized task execution improves Sprint's performance by 25.6% in the case of ES (see Figure 5).

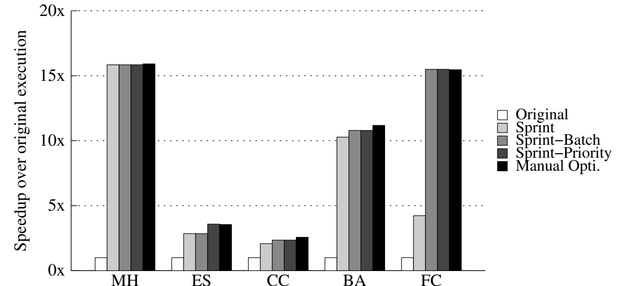

Figure 5. Speedup of automatically Sprint-ed execution over original. Benefits of batching and prioritized execution are also shown. Sprint's performance gains are comparable to the best manual optimizations which include both asynchronous and batch query execution but avoid Sprint's duplicate computation overhead.

6.2 Program Characteristics

Two fundamental program characteristics limit speedup:

1. Ratio of remote data access latency and computation latency: The higher the amount of time a program spends in remote data accesses compared to time spent in "computing the addresses of the accesses" and other operations, the more scope Sprint has to improve program execution.
2. Length of remote data access dependency chains: The length of the longest dependency chain times a single remote data access latency is the lower bound on the time to execute all the remote data accesses in a program. Table 3 shows the length of the longest chain for each program.

Assuming infinite resources and perfect scalability of the remote data source, Table 3 shows the theoretical upper bound on achievable speedup.

6.3 Cache Behavior

Table 4 shows the number of cache hits, misses, and waits (requests that had already been issued by O causing P to wait). Sprint converts a lot of remote accesses by the original program into local accesses to the cache in the Sprint-ed program.

Benchmark (Abbreviation)	Description	Algorithm	Data Source	Input Size (in number of remote accesses)
Management Hierarchy (MH)	Builds manager-employee relationship of organization	Tree Building	YP	766
Employee Search (ES)	Finds employee meeting certain search parameters	Tree Search	YP	293
Citations Count (CC)	Aggregates citations of a research group	Tree Traversal	DB2	502
Bibliography Aggregation (BA)	Aggregates bibliography under each manager in research organization	Tree Building, Traversal	YP, DB2	1268
Friend Connectivity (FC)	Displays connectivity of Facebook friends circle	Transitive Closure of Graph	FB	401

Table 2. Benchmark details

Benchmark	Remote Access Time (Percentage of Total)	Longest Dependence Chain Length	Upper bound on Speedup
MH	97.42%	4	31.36x
ES	97.17%	7	19.41x
CC	99.92%	2	209.16x
BA	98.27%	4	49.02x
FC	98.45%	2	48.99x

Table 3. Fraction of total execution time spent in remote data accesses and length of longest dependency chain limit the speedup achievable by Sprint.

Benchmark	Cache				
	Accesses	Hits	Waits	Misses	Miss %
MH	766	747	10	9	1.12%
ES	293	197	48	48	16.38%
CC	502	202	168	132	26.29%
BA	1268	949	178	141	11.12%
FC	401	394	0	7	1.75%

Table 4. Cache statistics: Waits denote accesses by the Pessimist that had already been issued by the Optimist but had not yet been serviced by the data source, causing the Pessimist to wait for the data.

Table 5 shows the distribution of useful and useless data prefetched by O. The ratio of useful prefetches to the total number of accesses shows that Sprint successfully prefetched a significant fraction of the used data. The cause of useless prefetches in Employee Search has already been discussed in Section 6.1 under *Prioritized Task Execution*.

Benchmark	Prefetches			
	Total	Useful	Useless	Useful %
MH	757	757	0	100.00%
ES	714	245	469	34.31%
CC	370	370	0	100.00%
BA	1127	1127	0	100.00%
FC	394	394	0	100.00%

Table 5. Cache statistics: Split of prefetches according to use by Pessimist. Sprint successfully prefetches a significant fraction of the used data.

6.4 Comparison with Sequential Prefetching

To understand the importance of parallel prefetching by many Optimist threads as opposed to sequential prefetching by a single thread, we implemented a sequential prefetcher and studied its performance using the FC (Facebook) application. Sequential prefetching yields 2% speedup over original program execution. By contrast, parallel prefetching unlocks an order of magnitude ($15\times$) speedup. The fundamental reason for the performance difference is: Sequential prefetching is limited by the time to process all nodes accessed in a data structure (work). Therefore, sequential prefetching can reduce overall latency by at most a constant factor of the total work. Parallel prefetching is limited by the time to process the nodes in the longest dependency chain (depth), which can be significantly smaller than the total work. Parallel prefetching exploits the parallelism inherent in the processing of different parts of a data structure.

7. Related Work

Batching The idea of batching is to convert several round trips into one, and thereby amortize the round trip cost over more data. Related remote data access calls are not performed at the point the client requests them, but are instead deferred until the client actually needs the value of a result. By that time, a number of deferred calls may have accumulated and the calls are sent all at once, in a "batch" [2, 14, 15]. The major disadvantage of these batching proposals is that they require the programmer to rewrite the code (both client-side and server-side) in non-trivial ways that typically obscure the program logic.

Parallelization Parallelization exposes independent remote data accesses and overlaps their round trip latencies. Manual parallelization using locks is error prone. Transactional memories with "abstract locking" could be used to simplify the task of parallelization [24]. However, both approaches often require the relaxation of the original semantics of the program. For example, assume that the for-loop on lines 15–21 in Listing 1 is parallelized. Synchronized updates to the `root` node on line 18 do not guarantee that the original program order is preserved. Unordered iterators in Galois have the same problem [19]. Ordered iterators could be used but that would serialize execution. Further, Galois still requires the programmer to implement synchronized access to data structures. Speculative parallelization systems such as Safe Futures [33] and Spice [27] are fully automatic; unfortunately, the memory dependencies between iterations cause iterations to be re-executed thereby resulting in the re-execution of the expensive remote data accesses. In contrast to thread-level speculation schemes, Sprint does not monitor all memory reads and writes and does not abort on all conflicts. Finally, none of these techniques incorporate batch execution where possible.

Program Slicing Conceptually, program slicing could be used to determine the slice of the program that is required to execute remote data accesses [13, 30]; this slice could be executed ahead of the remainder of the program thereby overlapping remote data communication with computation. Furthermore, automatic parallelization tools could be used to optimize the "prefetcher slice". However, automatic slicing and parallelization tools rely on analyses (such as pointer analysis) being interprocedural. These analyses are typically imprecise for complex code. Furthermore, these analyses may have to be applied at the bytecode or binary level when recompilation of source code is not an option. Advances in the program slicing front could reduce the amount of duplicate computation performed by a Sprint-ed program.

Memory Prefetching To mask the latency of servicing memory operations that miss in the cache, prefetching via pre-execution has been proposed. This approach uses compiler analyses to generate a backward slice ("p-slice") from the address of each target memory operation. All stores in

the backward slice are elided. A p-slice is scheduled for execution by a helper thread that acts as a prefetcher for a delinquent load [32, 35]. Like the prefetching approach proposed by Chang and Gibson [4], the elimination of flow dependencies from stores to loads causes divergence between the main thread and the helper thread, requiring frequent synchronization. In contrast to p-slices, Sprint generates Optimist threads that are long running and without store elisions thereby respecting most dependencies and avoiding resynchronization costs.

Lee et al. propose a helper thread construction algorithm that privatizes only array locations that are written in the backward slices from the addresses of the target memory operations, and constructs a single helper thread [20]. Sprint handles both arrays and records, and constructs multiple helper threads. Additionally, the technique proposed by Lee et al. is restricted to regions with a single live-in; Sprint does not have such a restriction.

Cooksey et al. propose content-based prefetching, a technique that scans data in cache lines to identify addresses and issues prefetch requests for those addresses [7]. By contrast, Sprint pre-computes addresses that will most likely be accessed—this approach is more suitable for the case when the ratio of remote access latency and computation latency is high (as in the domain of interest). Whereas content-based prefetching traverses recursive data structures sequentially, Sprint's unique code transformation enables the prefetcher to traverse data structures in parallel. This gives Sprint a significant performance advantage.

Sprint's execution engine is unique to the application domain. Most memory prefetching techniques assume hardware support for fast pre-execution thread initialization, cache line scanning, etc., whereas Sprint runs programs on commodity systems. Another way of viewing memory prefetching is that it can serve to reduce the request processing latency at the data source by reducing last level cache misses, whereas Sprint reduces the total round trip network latency by overlapping the latencies of multiple data requests in the client. As mentioned in Section 6, the round trip latency (and not the processing time at the data source) was the dominant factor in the overall application latency for the application and data source combinations studied in this paper. To cover the space of diverse workloads, Sprint and memory prefetching can be combined in a complementary fashion.

I/O Prefetching To mask the latency of filesystem access, prefetching via history-based prediction [8, 18] and via pre-execution [4, 5, 34] has been used. As discussed in Section 1, history-based prefetchers work only for programs with regular data access patterns. When accesses lack regularity, as in the domain of interest, history-based prefetchers cannot help. SpecHint speculates future I/O accesses [4]. In the presence of dependencies between accesses, SpecHint often causes misspeculation of future disk accesses and spurious disk accesses. By contrast, Sprint generates Optimist threads that are long running without store elisions thereby respecting most dependencies and avoiding resynchronization costs. Other pre-execution approaches [5, 34] rely on slicing techniques similar to the one used to construct memory prefetchers, and thus suffer from the problems discussed earlier. Sprint uses speculative parallelization to avoid these shortcomings. In those pre-execution approaches, program slicing is used to construct a single prefetcher thread per main thread. Sprint's code transformation creates several useful parallel prefetcher threads per main thread that can yield significantly better performance. Patterson and Gibson modify programs to insert prefetch hints to the filesystem [26]. Koller and Rangaswami propose I/O deduplication that reduces duplication of data on disk by introducing a content addressable cache that is indexed on *write* operations to avoid writing duplicate data and to improve write latency; read operations are cached based on history [17]. By contrast, Sprint is designed to reduce the latency of *read* operations that are not amenable to history-based caching and prefetching.

Web Prefetching Prefetching based on cumulative usage statistics at the client and server of pages linked via hyperlinks in a webpage has been proposed [10, 25]. The shortcomings of history-based approaches have already been discussed (Section 1). Mickens et al. proposed Crom, a JavaScript speculation engine to accelerate rich web applications [22]. Sprint and Crom perform complementary speculation. "Crom speculates on user activity rather than results of data accesses" [22]. Sprint speculates on the results of data accesses. Referring back to the discussion on input operations in Section 5.2.2, Crom can speculate input events while Sprint can accelerate processing of each input event handler resulting in an expected multiplicative performance improvement. Eden et al. propose to modify the browser to enable a user to indicate future webpage accesses via a click [11]. The browser then prefetches the page so that the page is available by the time the user navigates to the page. Sprint is entirely transparent to the user.

8. Conclusion

Many modern programs spend a significant portion of their execution time waiting for data from remote data sources. Sprint automatically reduces the total latency of such programs while preserving their semantics. Sprint provides speedups between 2.4× to 15.8× on a set of applications that access different data sources. Sprint combines both parallel and batch execution of remote data accesses. Sprint extends the state-of-the-art in prefetching for irregular and input-dependent data access patterns. Indeed, the techniques presented here are applicable in other contexts such as prefetching from disk. Since the system is specified in terms of a data access API, porting Sprint to other applicable contexts is just a matter of specifying the relevant API in that context.

Acknowledgments

We thank Mark Wegman and Nick Mitchell for their inputs during various stages of this work. We also thank the anonymous reviewers for their insightful feedback.

References

[1] M. Allen. RestFB: Facebook Java api. http://restfb.com.

[2] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proc. of OOPSLA '94*.

[3] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and Extensible Component Systems*, 2002.

[4] F. Chang and G. A. Gibson. Automatic I/O hint generation through speculative execution. In *Proc. of OSDI '99*.

[5] Y. Chen, S. Byna, X.-H. Sun, R. Thakur, and W. Gropp. Hiding I/O latency with pre-execution prefetching for parallel applications. In *Proc. of SC '08*.

[6] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *Proc. of MICRO '01*.

[7] R. Cooksey, S. Jourdan, and D. Grunwald. A stateless, content-directed data prefetching mechanism. In *Proc. of ASPLOS '02*.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *Proc. of SIGMOD '93*.

[9] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proc. of PLDI '07*.

[10] D. Duchamp. Prefetching hyperlinks. In *Proc. of USENIX '99*.

[11] A. N. Eden, B. W. Joh, and T. Mudge. Web latency reduction via client-side prefetching. In *Proc. of ISPASS '00*.

[12] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proc. of PASTE '10*.

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proc. of PLDI '88*.

[14] A. Ibrahim, M. F. II, W. R. Cook, and E. Tilevich. Remote batch invocation for web services: Document-oriented web services with object-oriented interfaces. In *Proc. of ECOWS '09*, .

[15] A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In *Proc. of ECOOP '06*, .

[16] D. Kim and D. Yeung. Design and evaluation of compiler algorithms for pre-execution. In *Proc. of ASPLOS '02*.

[17] R. Koller and R. Rangaswami. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage (TOS)*, 2010.

[18] D. Kotz and C. S. Ellis. Practical prefetching techniques for parallel file systems. In *Proc. of PDIS '91*.

[19] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proc. of PLDI '07*.

[20] J. Lee, C. Jung, D. Lim, and Y. Solihin. Prefetching with helper threads for loosely coupled multiprocessor systems. *IEEE Transactions on Parallel and Distributed Systems*, 2009.

[21] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. *SIGARCH Computer Architecture News*, 2001.

[22] J. Mickens, J. Elson, J. Howell, and J. Lorch. Crom: Faster web browsing using speculative execution. In *Proc. of NSDI '10*.

[23] T. C. Mowry, A. K. Demke, and O. Krieger. Automatic compiler-inserted I/O prefetching for out-of-core applications. In *Proc. of OSDI '96*.

[24] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proc. of PPoPP '07*.

[25] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, 1996.

[26] R. H. Patterson and G. A. Gibson. Exposing I/O concurrency with informed prefetching. In *Proc. of PDIS '94*.

[27] E. Raman, N. Vachharajani, R. Rangan, and D. I. August. Spice: speculative parallel iteration chunk execution. In *Proc. of CGO '08*.

[28] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *Proc. of VLDB '06*.

[29] J. M. Tamayo. Pepe JDBC dependency tracker. http://github.com/jtamayo/pepe.

[30] F. Tip. A survey of program slicing techniques. Technical report, CWI, Amsterdam, The Netherlands, The Netherlands, 1994.

[31] K. S. Trivedi. On the paging performance of array algorithms. *IEEE Transactions on Computers*, 1977.

[32] P. H. Wang, J. D. Collins, H. Wang, D. Kim, B. Greene, K. ming Chan, A. B. Yunus, T. Sych, S. F. Moore, and J. P. Shen. Helper threads via virtual multithreading on an experimental Itanium 2 processor-based platform. In *Proc. of ASPLOS '04*.

[33] A. Welc, S. Jagannathan, and A. Hosking. Safe Futures for Java. In *Proc. of OOPSLA '05*.

[34] C.-K. Yang, T. Mitra, and T.-c. Chiueh. A decoupled architecture for application-specific file prefetching. In *Proc. of FREENIX Track: USENIX '02*.

[35] W. Zhang, D. M. Tullsen, and B. Calder. Accelerating and adapting precomputation threads for effcient prefetching. In *Proc. of HPCA '07*.