Unsynchronized Techniques for Approximate Parallel Computing

Martin C. Rinard
MIT EECS and CSAIL
rinard@csail.mit.edu

Abstract

We present techniques for obtaining acceptable unsynchronized parallel computations. Even though these techniques may generate interactions (such as data races) that the current rigid value system condemns as incorrect, they are engineered to 1) preserve key data structure consistency constraints while 2) producing a result that is accurate enough, often enough. And because they contain no synchronization, they eliminate the synchronization overhead, parallelism reduction, and failure propagation typically associated with synchronization mechanisms.

We also identify a class of computations that interact well with our techniques. These computations combine multiple contributions to obtain a composite result. Redundancy in the contributions enables the unsynchronized computation to produce an acceptably accurate result even though it may, in effect, drop contributions. Thus our unsynchronized techniques (either directly or indirectly) are often acceptably accurate for the same reason as other successful approximate computing techniques such as task skipping and loop perforation.

1. Introduction

To eliminate potentially undesirable interactions (such as data races) that may occur when multiple parallel threads access shared data, developers often augment parallel programs with synchronization. The disadvantages of standard synchronization mechanisms include:

- **Synchronization Overhead:** The time overhead of executing the synchronization primitives and the space overhead of the state required to implement the primitives.
- **Parallelism Reduction:** Synchronization can reduce parallelism by forcing one or more threads to wait for other threads.
- **Failure Propagation:** With standard synchronization mechanisms, threads often wait for another thread to perform a synchronization operation (such as releasing a lock or hitting a barrier) before they can proceed. If the thread does not perform the operation (because of an infinite loop, a crash, or a coding error that omits the operation), the waiting threads can hang indefinitely.

Now, it may not be immediately clear that it is possible to eliminate potentially undesirable interactions in parallel computations without incurring one or more of these disadvantages. We don't even try — we instead develop approaches in which such interactions may indeed take place as long as they occur infrequently enough for the computation to produce an acceptably accurate result with high enough likelihood. In other words, instead of attempting to deliver a perfect/correct computation, we only aspire to deliver a computation that is *accurate enough, often enough*.

This change in perspective calls into question the legitimacy of the current rigid value system, which classifies all data races as undesirable [4]. It also opens up new regions of the engineering space and enables us to develop new mechanisms and techniques that deliver advantages unobtainable with previous approaches. Armed with this new perspective, we develop the following concepts, mechanisms, and techniques:

- **Integrity and Accuracy Instead of Correctness:** The field of program analysis and verification has traditionally focused on obtaining correct programs that always satisfy traditional correctness properties. With this approach, each program is characterized either as correct or incorrect.

  We see this classification as counterproductive — there are many incorrect programs that provide worthwhile results for their users. In fact, it is common knowledge that essentially every large software system has errors and is therefore incorrect. One may very reasonably question the utility of a conceptual framework that condemns essentially every successfully deployed artifact as incorrect.

  We therefore focus instead on acceptability, in particular two kinds of acceptability: *integrity* and *accuracy*. Integrity constraints capture properties that the program must satisfy to execute successfully to produce a well-formed result. Accuracy constraints characterize how accurate the result must be. To date we have largely focused on obtaining programs that always satisfy their integrity constraints but only satisfy their accuracy constraints with high likelihood. But we anticipate that other combinations (for example, programs that satisfy the in-

tegrity constraints only with high likelihood) will be appropriate for many usage contexts.

- **Semantic Data Races:** Data races have traditionally been defined in terms of unsynchronized reads and writes to shared data [4]. This approach misses the point in at least two ways: 1) the absence of data races does not ensure that the computation satisfies any particular useful property (note that inserting a lock aquire and release around every access to shared data eliminates data races but does not change the semantics) and 2) the presence of data races does not ensure that the computation has any undesirable properties. The problem is that the current concept of data races focuses on a narrow aspect of the program's internal execution that may or may not be relevant to its semantics.

  We instead propose to characterize interactions in terms of the acceptability properties of the shared data that they may or may not preserve. If an interaction violates a specified property, we characterize the interaction as a semantic data race with respect to the property that it may violate. With this perspective, synchronization becomes necessary only to the extent that it is required to preserve essential acceptability properties.

  Prioritizing different acceptability properties makes it possible to classify potential interactions between parallel threads according to the properties that they may or may not violate. This characterization, in turn, induces a trade-off space for synchronization mechanisms. An expensive synchronization mechanism, for example, may preserve most or all of the properties. A less expensive synchronization mechanism, on the other hand, may preserve fewer properties. The trade off between the desirability of the properties and the cost of the synchronization mechanisms required to preserve these properties determines the best synchronization mechanism for any given context.

- **Synchronization-Free Updates to Shared Data:** Developers have traditionally used mutual exclusion synchronization to make updates to shared data execute atomically. We have identified a class of data structures and updates that can tolerate unsynchronized updates (even when the lack of synchronization may cause updates from different parallel threads to interfere). These data structures and updates are characterized by:

  - **No Synchronization, No Atomicity:** The updates operate with no synchronization and provide no atomicity guarantees. There is therefore no synchronization overhead, no parallelism reduction, and parallel threads continue to execute regardless of what happens in other threads.
  - **Preservation of Key Consistency Properties:** The data structures come with a set of standard consistency properties. The data structure clients, however, can often use the data structure successfully even if some of these consistency properties do not hold. Even though the unsynchronized updates may violate some of these properties, they preserve the key properties that the clients require to execute successfully (i.e., without crashing to produce a result that is accurate enough, often enough).

    The updates we consider in this paper have the property that even if an update may perform multiple writes to shared data, each individual write preserves the key consistency properties. This feature ensures that these properties hold in all possible executions. In general, however, we believe it will also be productive to consider updates that use data structure repair to repair temporary violations of the key consistency properties or updates that ensure the key properties hold only with high likelihood.
  - **Acceptably Infrequent Interactions:** In theory, it may be possible for non-atomic interactions to occur frequently enough to produce unacceptably accurate results. In practice, however, such potentially undesirable interactions occur infrequently enough so that the result is acceptably accurate.
  - **Internal Integrity:** The unsynchronized updates never crash or perform an illegal operation such as an out of bounds access.

- **Barrier Elimination:** We also discuss a mechanism that eliminates barrier idling — no thread ever waits at a barrier [23]. The mechanism is simple. The computation contains a set of threads that are cooperating to execute a group of tasks that comprise a parallel computation phase. The standard barrier synchronization mechanism would force all threads to wait until all tasks have been completed.

  Our alternate mechanism instead has all threads proceed on to the next phase (potentially dropping the task they are working on) as soon as there are too few tasks to keep all threads busy. The net effect is 1) to eliminate barrier idling (when threads wait at a barrier for other threads to finish executing their tasks) by 2) dropping or delaying a subset of the tasks in the computational phase to produce 3) an approximate but accurate enough result.

We note that both our synchronization-free data structures and barrier elimination mechanisms exploit the fact that many programs can produce acceptably accurate results even though they execute only a subset of the originally specified computation. This property is the key to the success of such fundamental approximate computing techniques as task skipping [22, 23] and loop perforation [15, 16, 25, 27]. Our synchronization elimination mechanisms deliver another form of approximate computing.

2. Unsynchronized Data Structure Updates

We next present several data structures with corresponding synchronization-free updates. We characterize the effect of these updates according to the consistency properties they are guaranteed to preserve and the consistency properties their interactions may violate. We identify two kinds of consistency properties:

- **History-Sensitive:** Consistency properties that depend on the history of operations applied to the data structure. For example, a history-sensitive property of a binary search tree might state that each element inserted into the tree is present in the tree.
- **Instantaneous:** Consistency properties that depend only on the current state of the data structure. For example, an instantaneous property of a binary search tree might state that all elements to the left of a given element are less than that element, while all elements to the right are greater than the element.

We note that there is a technical detail associated with data structure consistency properties in the presence of parallelism — namely, that the consistency properties are most naturally stated with respect to a quiescent data structure state in which there are not ongoing operations. In a parallel computation such a state may never exist (i.e., the data structure may always have at least one ongoing operation). Strictly speaking, it may therefore be necessary to reason about data structure states that may result if all operations are allowed to complete with no new operations allowed to start.

2.1 An Accumulator

We start by considering a simple accumulator with a single `add` operation:

```
double accumulator;

void add(double value) {
  accumulator = accumulator + value;
}
```

The `add` operation reads the `accumulator`, adds `value`, then stores the sum back into the `accumulator`. Note that if two threads simultaneously perform an `add` operation, it is possible for both threads to read the `accumulator`, add their `values`, then write the sum back into the `accumulator`. The net result is that the accumulator includes only one, but not both, of the `values`. This has traditionally been considered to be a classical data race and concurrency error.

From our perspective, this interaction violates a history-sensitive property, namely that the `accumulator` include all of the `values` from the `add` operations performed on it in the past. The interaction is therefore a semantic data race because it violates this history-sensitive property.

But on machines with atomic reads and writes, the interaction does not violate the instantaneous property that `accumulator` contains a number. If the client of the accumulator does not require the sum to be exact, and if the interaction above occurs infrequently enough, the unsynchronized `add` operation presented above may provide an acceptably accurate result (even if the `accumulator` does not contain some of the added `values`) [13–15].

This example illustrates a basic pattern that is repeated in more complex data structures such as lists, trees, and graphs. Many data structures contain operations that conceptually add elements into the data structure. It is often possible to obtain unsynchronized updates that may drop added elements (thereby violating the history-sensitive property that the data structure contains all of the added elements) but otherwise leaves the data structure consistent (i.e., the data structure respects all instantaneous consistency properties).

For example, standard sequential (unsynchronized) implementations of operations that add elements to linked list and binary search tree data structures, when run in parallel without synchronization, may drop elements but preserve the instantaneous consistency properties of the data structures.

2.2 A Growable Array

We next consider a growable array data structure with an `add` operation:

```
class array {
  public:
  double *values;
  int next;
  int length;
};

struct array *current;

void add(double value) {
  array *c = current;
  int l = c->last;
  int n = c->next;
  if (l <= n) {
    double *v = new double[l*2];
    for (int i = 0; i <= l; i++) {
      v[i] = c->values[i];
    }
    array *a = new array;
    a->values = v;
    a->length = l*2;
    a->next = n;
    current = a;
    c = a;
  }
  c->values[n] = value;
  c->next = n+1;
}
```

A key element of the acceptability of this algorithm in the presence of parallel unsynchronized updates is that each write preserves the instantaneous consistency properties of the underlying data structure. This element ensures that the data structure *always* preserves these consistency properties regardless of the way the parallel writes interleave. In this case the key consistency property is that `current->next` must always be less than `current->length`.

This key element ensures the internal integrity of the `add` operation. To avoid out of bounds accesses, it must always be the case that `n` is less than the length of `c->values` when the statement `c->values[n] = value` executes. We ensure that this property is true for every `array` that `current` references by setting `c->length` to the length of `c->values` when `c` is created (it is critical that this initialization takes place without interference from other threads), then never changing `c->length`.

Note that replacing the statement `c->values[n] = value` with `c->values[c->next] = value` produces an `add` operation that may violate internal integrity with an out of bounds access — it is possible for another `add` operation to change `c->next` in between the check to see if `c->next` is in bounds and the use of `c->next` to insert the next `value` into the array.

There are many ways this unsynchronized data structure can drop added `values`, for example:

- **Simultaneous Insertion:** Two `add` operations can retrieve the same `c->next` and insert their two `values` into the same array element.
- **Simultaneous Grow and Insertion:** Two `add` operations may determine that there is no room in the current array. Both then start to execute the code that grows the array. One finishes, other threads insert their `values` into the new array, then the second grow code finishes and installs a new version of the array without the inserted elements.

If these kinds of interactions occur infrequently enough, the dropped elements can have an acceptable effect on the accuracy.

2.3 Single Write Commits

We note that, conceptually, each modification to the growable array commits with a single write — either at the statement `current = a` (this statement commits the array grow operation) or at the statement `c->next = n` (this statement commits the insertion of `value` into the array). Unsynchronized updates with this property tend to work well — if modifications require multiple writes to commit simultaneously, it may be difficult to order the writes in such a way that each write preserves the relevant consistency properties. In this case, it may be possible for operations to interleave in a way that produces an inconsistent data structure with some modifications from one operation and others from another operation.

2.4 Final Check

We call the time between when the `add` operation decides to update the data structure and the time when the update has actually taken place the *window of vulnerability* (because this is the time when other operations can interfere and cause elements to be dropped). One way to make the window of vulnerability smaller is to recheck the original condition just before performing the update. If the original condition still holds, the update proceeds. Otherwise, the operation retries in the new state.

The following code uses a final check to make the window of vulnerability smaller:

```
class array {
  public:
  double *values;
  int next;
  int length;
};

struct array *current;

void add(double value) {
  array *c = current;
  int l = c->last;
  int n = c->next;
  while (l <= n) {
    double *v = new double[l*2];
    for (int i = 0; i <= l; i++) {
      v[i] = c->values[i];
    }
    array *a = new array;
    a->values = v;
    a->length = l*2;
    a->next = n;
    // final check
    if (current == c) {
      current = a;
      c = a;
    } else {
      c = current;
      l = c->last;
      n = c->next;
    }
  }
  c->values[n] = value;
  c->next = n+1;
}
```

2.5 Parallel Space Subdivision Tree Construction

We have used the ideas presented above, along with data structure repair, to obtain a synchronization-free parallel space subdivision tree construction algorithm for the Barnes-Hut N-body simulation algorithm [2, 21]. Eliminating synchronization improves the performance of the parallel algo-

rithm by approximately 20% with negligible impact on the overall accuracy of the simulation. The use of final checks to reduce the sizes of various windows of vulnerability significantly reduces the number of dropped bodies (each operation in the tree construction algorithm inserts a body into the space subdivision tree under construction).

3. Connections With Other Techniques

Task skipping [22, 23] and loop perforation [15, 16, 25, 27] provide benefits such as increased robustness and fault tolerance, improved performance and energy consumption, and eliminating idle time in parallel computations by dropping subcomputations.

Many data structures exist to collect contributions from multiple subcomputations, with each update integrating a contribution into the data structure. Interactions between unsynchronized updates may drop contributions, with same effect on the result as dropping the subcomputations that produced the contributions.

Other data structures exist to organize the computation, with each element in the data structure corresponding to one or more subcomputations. The elements in the Barnes-Hut space subdivision tree, for example, correspond to computations that calculate part of the force acting on each body in the simulation. If an unsynchronized update drops one or more elements, the net effect is often to drop the corresponding subcomputation.

Unsynchronized updates therefore often have, indirectly, the same effect that task skipping and loop perforation have directly — in some cases the unsynchronized update drops a contribution from a complete subcomputation, in others a dropped element corresponds to a dropped subcomputation. Task skipping, loop perforation, and unsynchronized updates therefore often share the same underlying reason behind their success.

It is also possible to exploit this phenomenon to obtain a parallelizing compiler that generates code with data races that may (acceptably) drop updates to shared data [13, 14]. Again, the reason is similar — dropping updates has the same end effect as dropping the subcomputation that generated the update.

We note that all of these mechanisms tend to work well with computations that generate many contributions, then combine the contributions to obtain a composite result. This broad category of computations includes many modern search, machine learning, financial, and scientific computations. Because of redundancy in the many contributions to the result, the computations can typically generate an acceptably accurate result with a subset of the contributions. Many of the computations that successfully tolerate loop perforation, task skipping, and unsynchronized updates tend to exhibit this pattern.

4. Performance Consequences

One of the primary performance consequences of unsynchronized updates is the elimination of the performance overhead from synchronization operations. The overall amortized performance impact depends on 1) how frequently the computation performs updates, and 2) how efficiently the underlying computational platform implements synchronization primitives.

The bottom line is that it should be possible to engineer a computational platform with negligible synchronization overhead. Whether vendors will choose to build such a platform depends on whether the performance benefits justify the engineering cost. In one plausible scenario, the cost of synchronization in typical systems will not justify maximal engineering effort — synchronization will be infrequent enough so that vendors will provide reasonably but not maximally efficient synchronization primitives (as is the case with current platforms). The following analysis assumes the computational platform is multithreaded and implements a shared address space (but similar issues arise on other platforms).

Mutual exclusion synchronization is typically tied to data access. In general, local data access is fast, while remote data access is slow. Standard shared address space caching protocols obtain exclusive local access to data when it is written (anticipating further accesses that should be made local for optimal performance). Similarly, synchronization primitives obtain local access to the synchronization state. A synchronized update to locally available data typically proceeds as follows: 1) a local synchronization operation (typically a lock acquire), 2) local accesses to shared data, then 3) a local synchronization operation (typically a lock release). Because the synchronization operations are local, they can be engineered to execute essentially as quickly as the local data accesses. So the overhead is determined by the size of the local data accesses (which are often large enough to profitably amortize the synchronization overhead) and how often the computation executes the update.

A synchronized update to remote data typically proceeds as follows: 1) a remote synchronization operation (which obtains exclusive local access to the synchronization state), 2) remote access to shared data (which obtains exclusive local access to the updated data structure state), and 3) a local synchronization operation. Once again, the synchronization overhead is determined by the size of shared data accesses and how often the computation executes the update.

It is possible to eliminate much of the synchronization overhead in the remote case by combining the synchronization operation with the remote access of both the synchronization state and shared data state. Optimistic synchronization primitives (such as load linked/store conditional) implement this combination linkage. It is also possible to imagine other mechanisms (such as prefetching the shared data state

along with the synchronization state at the first synchronization operation).

A final consideration is how efficiently the synchronization primitives are engineered. Because such primitives occur fairly infrequently in current and envisioned multithreaded computations, they tend to be less intensively engineered than more common primitives (such as loads, stores, and arithmetic primitives). In summary:

- It is possible to implement computational platforms that provide synchronization with little or no overhead.
- But current usage patterns and incentives do not support the engineering of such platforms.
- So there are likely to be some performance gains available from unsynchronized updates. We would expect the 20% performance gain we observed for the unsynchronized parallel Barnes-Hut space subdivision tree construction algorithm [21] to be generally representative of the benefits available for computations that perform frequent updates to potentially shared data.
- Of course, if the underlying synchronization primitives are not implemented reasonably efficiently (for example, if they perform a remote operation even when the shared data is available locally), then much more substantial gains should be possible.

5. Eliminating Barrier Synchronization

We now turn our attention to eliminating barrier synchronization. The basic idea is simple — ensure that threads can proceed beyond the barrier without waiting for other threads to continue. If the other threads drop their tasks, the effect is to terminate the computational phase (the barrier marks the end of this phase) early. If the other threads continue to execute, the phases overlap. To avoid having the threads become completely out of phase, it may be desirable to occasionally resynchronize.

We would expect this approach to work well when one phase produces data (typically by combining multiple contributions to a composite result) that the next phase consumes. In this case we again see the effect of the synchronization elimination is to drop some of the subcomputations. If the parallel tasks from adjacent phases overlap, the subcomputations are not dropped but are simply late. In this case the second phase may observe a fully computed result if the subcomputations arrive from the completion of tasks in the first phase by the time it accesses the result.

In most cases the computational phases are large enough to make the amortized overhead of the barrier operation itself negligible — the performance impact comes from load imbalances as threads wait at the barrier for other threads to complete and hit the barrier.

6. Related Work

Early work on synchronization elimination focused on *lock coarsening* — replacing repeated adjacent lock acquires and releases with a single acquire and release [7, 8, 10, 11], potentially with dynamic feedback to select the optimal lock granularity as the computation executes [9, 12]. The goal is to eliminate the overhead associated with the lock acquire and release primitives. The transformation may also reduce the available concurrency by increasing the size of the region over which the lock is held. A similar approach can reduce synchronization overhead in sequential executions of concurrent object-oriented programs [17]. Unlike the synchronization removal techniques discussed in this paper, lock coarsening preserves the semantics of the original program.

Early Java implementations had significant synchronization overhead. This overhead was caused by the combination of 1) many Java data structures were thread-safe and always synchronized their operations and 2) the Java implementation of the synchronization primitives was fairly inefficient. In response, researchers developed escape analysis algorithms that located objects accessed by only a single thread [1, 3, 5, 6, 24, 26]. The compiler then removed the synchronization otherwise associated with such objects. Java implementations have since evolved to have more efficient synchronization primitives and to give the developer explicit control over which data structures are synchronized.

Many of the computations that work well with unsynchronized updates and the elimination of barrier synchronization combine multiple contributions to obtain a result. In many cases it is possible to improve the performance of such updates by using optimistic synchronization primitives such as load linked/store conditional [18, 20]. It is also possible to detect heavily contended data by observing contention for the lock that makes updates to that data atomic. Threads then eliminate the contention by performing unsynchronized updates on local replicas, then combining the replicas at the end of the computational phase to obtain the result [13, 14, 19]. The elimination of remote accesses and synchronization overhead can make this technique extremely effective in enabling efficient parallel execution.

7. Conclusion

Synchronized parallel computations can suffer from synchronization overhead, parallelism reduction, and failure propagation when a thread fails to execute a synchronization operation. We present a set of unsynchronized techniques that eliminate these effects. These techniques are at the forefront of the emerging field of approximate computing, which exploits the freedom that many programs have to deliver results that are only accurate enough, often enough. The success of these techniques casts doubt on the legitimacy of the current rigid value system [4], which condemns these techniques as unsound and incorrect even though they produce successful executions.

References

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from Java programs. In *Proceedings of the 6th International Static Analysis Symposium*, September 1999.

[2] J. Barnes and P. Hut. A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324(4):446–449, 1986.

[3] B. Blanchet. Escape analysis for object oriented languages. application to Java. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.

[4] Hans-Juergen Boehm and Sarita V. Adve. You don't know jack about shared variables or memory models. *Commun. ACM*, 55(2):48–54, 2012.

[5] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.

[6] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.

[7] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. *Concurrency Practice & Experience*, 11(13):773–802, November 199.

[8] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Languages and Compilers for Parallel Computing, Ninth International Workshop*, pages 285–299, San Jose, CA, August 1996. Springer-Verlag.

[9] P. Diniz and M. Rinard. Dynamic feedback: An effective technique for adaptive computing. In *Proceedings of the SIGPLAN '97 Conference on Program Language Design and Implementation*, pages 71–84, Las Vegas, NV, June 1997.

[10] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 187–200, Paris, France, January 1997.

[11] P. Diniz and M. Rinard. Lock coarsening: eliminating lock overhead in automatically parallelized object-based programs. *Journal of Parallel and Distributed Computing*, 49(2):218–244, March 1998.

[12] P. Diniz and M. C. Rinard. Eliminating synchronization overhead in automatically parallelized programs using dynamic feedback. *ACM Transactions on Computer Systems*, 17(2):89–132, May 1999.

[13] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. *ACM Transactions on Embedded Computing Systems*. "to appear".

[14] S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, August 2010.

[15] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *SAS*, pages 316–333, 2011.

[16] Sasa Misailovic, Stelios Sidiroglou, Henry Hoffmann, and Martin C. Rinard. Quality of service profiling. In *ICSE (1)*, pages 25–34, 2010.

[17] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, California, USA, January 1995.

[18] M. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 112–123, Las Vegas, NV, June 1997.

[19] M. Rinard and P. Diniz. Eliminating synchronization bottlenecks in object-based programs using adaptive replication. In *1999 ACM International Conference on Supercomputing*, pages 83–92, Rhodes, Greece, June 1999.

[20] M. C. Rinard. Effective fine-grain synchronization for automatically parallelized programs using optimistic synchronization primitives. *ACM Transactions on Computer Systems*, 17(4):337–371, November 1999.

[21] Martin Rinard. A lossy, synchronization-free, race-full, but still acceptably accurate parallel space-subdivision tree construction algorithm. Technical Report MIT-CSAIL-TR-2012-005, MIT, February 20012.

[22] Martin C. Rinard. Probabilistic accuracy bounds for fault-tolerant computations that discard tasks. In *ICS*, pages 324–334, 2006.

[23] Martin C. Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *OOPSLA*, pages 369–386, 2007.

[24] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the 8th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Snowbird, Utah, June 2001.

[25] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *SIGSOFT FSE*, pages 124–134, 2011.

[26] J. Whaley and M. Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, Denver, CO, November 1999.

[27] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, pages 441–454, 2012.