Speculative Separation for Privatization and Reductions

Nick P. Johnson Hanjun Kim Prakash Prabhu Ayal Zaks† David I. August

Princeton University, Princeton, NJ
{npjohnso, hanjunk, pprabhu, august}@princeton.edu

†Intel Corporation, Haifa, Israel
ayal.zaks@intel.com

Abstract

Automatic parallelization is a promising strategy to improve application performance in the multicore era. However, common programming practices such as the reuse of data structures introduce artificial constraints that obstruct automatic parallelization. Privatization relieves these constraints by replicating data structures, thus enabling scalable parallelization. Prior privatization schemes are limited to arrays and scalar variables because they are sensitive to the layout of dynamic data structures. This work presents Privateer, the first fully automatic privatization system to handle dynamic and recursive data structures, even in languages with unrestricted pointers. To reduce sensitivity to memory layout, Privateer speculatively separates memory objects. Privateer's lightweight runtime system validates speculative separation and speculative privatization to ensure correct parallel execution. Privateer enables automatic parallelization of general-purpose C/C++ applications, yielding a geomean whole-program speedup of 11.4× over best sequential execution on 24 cores, while non-speculative parallelization yields only 0.93×.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming—Parallel Programming; D.3.4 [*Programming Languages*]: Processors—Compilers, Optimization

General Terms Languages, Performance, Design, Experimentation

Keywords Automatic parallelization, Separation, Speculation

1. Introduction

The microprocessor industry has committed to multicore architectures. These additional hardware resources, however, offer no benefit to sequential applications. Automatic parallelization is a promising approach to achieve performance on existing and new applications without additional programmer effort or application changes.

Yet automatic parallelization is not the norm. One limiting factor is the compiler's inability to distribute work across processors due to reuse of data structures. This reuse does not contribute to the constructive data flow of the program, but creates contention that prevents efficient parallel execution. A parallelizing compiler must either respect this contention by enforcing exclusivity on accesses to shared data structures, or ignore it and risk data races that change program behavior.

A compiler can remove contention by creating a *private* copy of the data structures for each worker process. Privatization eliminates contention and relaxes the program dependence structure by replicating the reused storage locations, producing multiple copies in memory that support independent, concurrent access. Similarly, reduction techniques relax ordering constraints on associative, commutative operators by replacing (or *expanding*) storage locations. Prior work [7, 21, 22, 29, 32] demonstrates that privatization and reductions are key enablers of parallelization.

Privatization Criterion	Memory Layout: Static	Memory Layout: Speculative
Speculative	LRPD [22] R-LRPD [7]	*Privateer (this work)*
Dynamic	PD [21]	
Static	Polaris [29] ASSA [14] Array Expansion [10] DSA [31] RSSA [23]	
Manual	Paralax [32]	STMs [8, 18]

Figure 1: Privatization Criterion and Memory Layout.

The applicability of privatization systems can be understood in two dimensions (Figure 1). A system uses the *Privatization Criterion* [21] to decide if replacing a shared data structure would change program behavior. To replicate object storage, a system determines *Memory Layout*: the base address and size of memory objects. Prior work assesses the privatization criterion statically [29], dynamically [21], or speculatively [7, 22].

However, prior work limits memory layout to arrays and scalar variables only and fails for programs that use linked or recursive data structures. The prevalent use of pointers and dynamic memory allocation creates a dichotomy between static accesses and objects. A pointer may refer to different objects of different sizes at different times, and static analysis usually fails to disambiguate these cases. As a result, it is difficult for a static compiler to decide *which objects* to duplicate, even when it can decide *which accesses* are private. Prior techniques are largely inapplicable to most C or C++ applications for this reason. Table 1 summarizes prior work.

This work proposes Privateer, the first fully automatic system capable of privatizing data structures in languages with pointers, type casts, and dynamic allocation. Instead of relying solely on static analysis to determine memory layout, Privateer employs profiling to characterize memory accesses at the granularity of memory objects. Using profiling information and static analysis, Privateer identifies accesses to memory objects that are expected to be iteration-private. Such objects are speculatively privatized, predicting that their accesses will remain iteration-private, thereby relaxing program dependence structure and enabling optimization and parallelization.

Privateer overcomes difficulties in memory layout while minimizing validation overheads. The loop's memory footprint is partitioned into several *logical heaps* according to observed access patterns. Privateer speculates that these heaps remain *separated* at runtime rather than speculating that individual memory access pairs are independent. Workers validate this separation property autonomously, requiring neither a log of accesses nor communication with other workers. This separation property is efficiently checked

PLDI'12, June 11–16, Beijing, China.

using compact metadata encoded in pointer *addresses*. Speculative separation reduces sensitivity to memory layout, thus allowing Privateer to extend an LRPD-style shadow memory test [22] to arbitrary objects. Privateer's robust, layout-insensitive privatization and reductions enable automatic parallelization of applications with linked and recursive data structures. This work contributes:

- the first **fully automatic** system to support privatization and reductions involving **pointers and dynamic allocation**;
- an efficient, scalable validation scheme for **speculative privatization and reductions** based on the **speculative separation** of logical heaps according to usage patterns; and,
- an application of this speculative privatization and reduction technique to the problem of **automatic parallelization**.

Privateer's transformations facilitate scalable automatic parallelization on commodity shared-memory machines. No programmer hints are used, nor are any hardware extensions assumed. We implement Privateer and evaluate it on a set of 5 programs. On a 24-core machine, results demonstrate a geomean whole-program speedup of 11.4× over best sequential execution. To achieve these results, Privateer privatized linked and recursive data structures which are beyond the abilities of prior work. Speculation via heap separation allows Privateer to extract scalable performance from general purpose, irregular applications.

2. Motivation

Automatic parallelization is sensitive to the dependences in a program. A single dependence may prevent a compiler from parallelizing an entire loop. Some dependences may never manifest, yet static analysis is unable to prove so. Speculation allows a compiler to overcome many of the limitations of static analysis. Instead of optimizing for a conservative worst case, a speculative compilation system assumes some common case of program behavior and optimizes accordingly [17, 19, 30]. A speculative system inserts code to validate these assumptions at runtime and recover when they fail. *Dependence speculation* is the application of speculative methods to remove those dependences which inhibit parallelization. However, dependence speculation is inappropriate for dependences which occur frequently. Privatization targets *false* (anti- and output-) dependences. Privatization succeeds even when false dependences are *frequent*, where dependence speculation fails.

Consider the code in Figure 2a (simplified from MiBench `dijkstra` [12]). Attempts to parallelize it are inhibited by frequent false dependences incident on reused data structures. The outer loop (Line 46) repeatedly performs Dijkstra's algorithm. However, the loop reuses two data structures across iterations: `Q`, a linked-list work queue (Line 5), and `pathcost`, an array of shortest path costs (Line 6). Although each iteration is conceptually independent, the reuse of `Q` and `pathcost` creates false dependences that impose an order on outer loop iterations, preventing parallelization.

These false dependences occur between every pair of iterations of the loop. If a naïve compiler were to speculate that these false dependences never manifest, the program would misspeculate on *every* iteration, and would fail to achieve scalable performance.

A privatization strategy is more appropriate for such cases. Privatization eliminates false dependences by creating a disjoint copy of the loop's memory footprint for each worker, enabling workers to proceed independently and without synchronization. Each worker operates on a different `Q` containing different linked list nodes and on different `pathcost` arrays. To prove the privatization criterion, the strategy must confirm the absence of a loop-carried flow dependence on every pointer load and store. This has been addressed using static analysis [29], dynamic tests [21], and speculation [7, 22] in prior work. To replace these data structures, the privatization strategy must also determine the memory layout.

Determining the memory layout entails identifying all private objects: `Q`, `pathcost`, and all linked list nodes. The memory layout enables the system to duplicate objects and re-route memory accesses so workers refer to their private copy. In the absence of pointers, a variable's source-level name uniquely identifies a memory object, allowing the compiler to determine the object's address and size. However, languages with pointers and dynamic allocation allow a many-to-many relationship between names and objects. Pointers refer to different objects at different times and allocation sites produce many objects, causing the code to exhibit different reuse patterns. Unlike related work, Privateer addresses the complications of pointers, type casts, and dynamic allocation.

Privateer speculatively separates the program state into several *logical heaps* according to the reuse patterns observed during profiling. The compiler indicates this separation to the runtime system, which in turn privatizes without concern for individual objects. By grouping objects, a logical heap can be privatized as a whole by adjusting virtual page tables, neither requiring complicated bookkeeping nor adjusting object addresses in a running program. All objects of each logical heap are placed within a fixed memory address range, allowing efficient validation of the separation property. In the example, `Q`, `pathcost`, and all linked list nodes are accessed privately whereas `adj` is only read; Privateer allocates them to distinct *logical heaps* of private objects and read-only objects at compile time, validating this separation at runtime.

Speculative separation greatly simplifies the memory layout problem, condensing unboundedly many objects into few heaps. This allows Privateer to apply a speculative privatization and reduction transformation on programs with pointers, dynamic allocation, and type casts. This removes false dependences, relaxing program constraints, and enables scalable automatic parallelization.

3. Design

Privateer is a combined compiler-runtime system that privatizes dynamic memory objects efficiently and addresses several challenges outlined in this section. The compiler system acts fully automatically without any guidance from the programmer. The compiler overcomes the limitations of static analysis by using profiling information to guide its transformations and produces code which interacts with the runtime system. The runtime system provides efficient mechanisms for replication of objects to support privatization and for recovery from misspeculation.

Privateer's privatization criterion, forbids cross-iteration flow dependences but, unlike [22], is not limited to arrays:

Privatization Criterion: *Let O be a memory object that is accessed in a loop L. O can be privatized if and only if no read from O returns a value written in an earlier iteration of L.*

Privateer also supports a related type of privatization that involves reduction operations with real flow dependences. The accumulator variable is expanded into multiple copies, each updated independently across iterations of the loop, after which all copies are merged to the final result. We list here our reduction criterion:

Reduction Criterion: *Let O be a memory object that is accessed in a loop L. O can be reduction-privatized if and only if all updates to O within L are performed by a single associative and commutative (reduction) operator, and no operation within L reads an intermediate value from O.*

The use of pointers and dynamic allocation in general purpose programs requires privatization and reduction systems to address:

1. *Rich Heap Model:* the solution needs to accurately distinguish among the many and diverse objects of the program, even when several objects are created by one static instruction.

Technique	Fully Automatic	Supports Pointers and Dynamic Allocation	Privatization			Reductions		
			Supported	Not limited by Static Analysis		Supported	Not limited by Static Analysis	
				Criterion	Memory Layout		Criterion	Memory Layout
Paralax [32]	×	-	✓	-	-	-	-	-
TL2 [8], Intel STM [18]	×	-	✓	-	-	-	-	-
PD [21], LRPD [22], R-LRPD [7]	✓	×	✓	✓	×	✓	✓	×
Hybrid Analysis [24]	✓	×	✓	✓	×	✓	✓	×
Array Expansion [10], ASSA [14], DSA [31]	✓	×	✓	×	×	×	-	-
STMLite+LLVM [17]	✓	✓	✓	✓	-	✓	×	×
CorD+Objects [27]	✓	✓	✓	×	×	✓	×	×
Privateer (this work)	✓	✓	✓	✓	✓	✓	✓	✓

Table 1: Comparison of Privateer with privatization and reduction schemes.

```
1  struct node {int vx; node *next}
2  struct queue {node *head,*tail}
3
5  queue Q;
6  int pathcost[N];
7  int adj[N][N];
8
9  void enqueueQ(int v) {
11  node* N = (node*)malloc(
12            sizeof(node));
13  N->vx = v;
16  N->next = Q.tail;
17  ...
20  Q.tail = N;
21 }
22
23 int dequeueQ(void) {
24  ...
27  qKill = Q.head;
30  v = qKill->vx;
33  Q.head = qKill->next;
35  free(qKill);
36  return v;
37 }
45 void hot_loop(int K) {
46  for (src=0; src < N; ++src) {
55   for (i=0; i < N; ++i)
56    pathcost[i] = infinity;
58
59   pathcost[src] = 0;
60   enqueueQ(src);
61
62   while (!emptyQ()) {
63    v = dequeueQ();
66    d = pathcost[v];
67    for (i=0; i < N; ++i) {
68     ncost = adj[v][i] + d;
69     if (pathcost[i] > ncost) {
73      pathcost[i] = ncost;
74      enqueueQ(i);
75     }
76    }
77   }
81  }
82 }
```

(a) Sequential `dijkstra` example.

```
1  struct node {int vx; node *next }
2  struct queue {node *head,*tail }
3
4  // Reallocation (Section 4.4)
5  queue *Q;
6  int *pathcost;
7  int **adj;
8
9  void enqueueQ(int v) {
10  // Reallocation (Section 4.4)
11  node* N = h_alloc(sizeof(node),
12                    SHORTLIVED);
13  N->vx = v;
14  // Privacy Check (Section 4.6)
15  private_read(&Q->tail, sizeof(node*));
16  N->next = Q->tail;
17  ...
18  // Privacy Check (Section 4.6)
19  private_write(&Q->tail, sizeof(node*));
20  Q->tail = N;
21 }
22
23 int dequeueQ(void) {
24  ...
25  // Privacy Check (Section 4.6)
26  private_read( &Q->head );
27  qKill = Q->head;
28  // Separation Check (Section 4.5)
29  check_heap( qKill, SHORTLIVED );
30  v = qKill->vx;
31  // Privacy Check (Section 4.6)
32  private_write( &Q->tail );
33  Q->head = qKill->next;
34  // Reallocation (Section 4.4)
35  h_dealloc(qKill, SHORTLIVED);
36  return v;
37 }
38
39 // Reallocation (Section 4.4)
40 void before_main(void) {
41  Q = h_alloc(sizeof(queue), PRIVATE);
42  pathcost = h_alloc(N*sizeof(int),PRIVATE);
43  adj = h_alloc(N*N*sizeof(int), READONLY);
44 }
45 void hot_loop(int K) {
46  for (src=0; src < N; ++src) {
47   // Privacy Check (Section 4.6)
48   private_write(&Q->head,sizeof(node*));
49   private_write(&Q->tail,sizeof(node*));
50   // Value prediction
51   Q->head = NULL;
52   Q->tail = NULL;
53   // Privacy Check (Section 4.6)
54   private_write(pathcost,N*sizeof(int));
55   for (i=0; i < N; ++i)
56    pathcost[i] = infinity;
57   // Privacy Check (Section 4.6)
58   private_write(&pathcost[src],sizeof(int));
59   pathcost[src] = 0;
60   enqueueQ(src);
61
62   while (!emptyQ()) {
63    v = dequeueQ();
64    // Privacy Check (Section 4.6)
65    private_read(&pathcost[v],sizeof(int));
66    d = pathcost[v];
67    for (i=0; i < N; ++i) {
68     ncost = adj[v][i] + d;
69     if (pathcost[i] > ncost) {
70      // Privacy Check (Section 4.6)
71      private_write( &pathcost[i],
72              sizeof(int) );
73      pathcost[i] = ncost;
74      enqueueQ(i);
75     }
76    }
77   }
78   // Value prediction
79   if( Q->head != NULL ) misspec();
80   if( Q->tail != NULL ) misspec();
81  }
82 }
```

(b) Speculatively privatized code, before parallelization. Changes are in grey.

Figure 2: Motivating example for Privateer. The original sequential application is on the left. The right shows the code after the speculative privatization transformation, before it is automatically parallelized. Unchanged lines are consistently numbered between (a) and (b).

2. *Robust Points-to Map:* Instructions manipulate pointers, yet privatization replicates memory objects. A robust mapping from pointers to objects is needed to consistently update both. In Figure 2, the system must determine the target object of pointers loaded from `queue` and `node` objects (Lines 27 and 33).
3. *Object Base, Size, Count:* The duplicated storage must be at least as large as the original. This also affects the allocation of metadata at per-object or per-byte resolution. In Figure 2, the system should know how many times, where, and how large it allocates before privatizing `node N` in Line 11.
4. *Replacement Transparency:* When replacing the storage in a running system, all pointers must remain valid. The system cannot *move* privatized storage as it cannot guarantee that all references will be updated. The system cannot even assume that pointer values are visible in the IR because of the possibility of "disguised" pointers [3].

Privateer overcomes these issues by speculating separation properties of the program. In this paper, we say that an *access path* is a sequence of operations which computes a pointer address, and that two access paths are *separated* if the sets of objects they name are disjoint. Separation is weaker than *points-to* information, since it does not enumerate the objects referenced by an access path. Separation is weaker than *alias* information, since it says nothing about two addresses within the same object. Yet separation information is strong enough to simplify memory layout. Further, separation can be validated at runtime without inter-worker communication.

3.1 Analysis and Transformation

The first task of the compiler is to recognize situations where privatization eliminates false dependences (and flow dependences for reductions), thereby enabling automatic parallelization. Privateer builds a representation of the dependence structure of the program's hot loops, and then employs memory and control profiling to remove rare and nonexistent dependences. This representation contains only frequently occurring dependences. Privateer interprets this as an optimistic view of the expected program dependences.

When the compiler discovers a hot loop that cannot be parallelized due to false or flow dependences, it investigates whether the privatization and reduction criteria apply. Privateer classifies every memory object according to its observed usage pattern within the loop. Based on this classification, the compiler decides if privatization is applicable and would enable parallelization.

The second task of the compiler is to perform the privatization transformation by inserting additional instructions to the program. These instructions interact with the runtime system to control the allocation of objects in memory and to validate that memory accesses match the expected patterns. The resulting speculatively privatized program is then amenable to automatic parallelization by parallelizing transformations such as DOALL.

3.2 Runtime Support System

Traditional privatization systems do not privatize many classes of dynamically allocated data structures since they are unable to determine the object sizes, number, and locations. Privateer takes a different approach. Privateer assigns each memory object to one of several *logical heaps*. At runtime, those objects are allocated within a known, fixed range for each logical heap. This simplifies the memory layout problem, since the runtime may treat each logical heap as a single object with known base and bound instead of many unknown objects. Privateer may test whether a pointer address falls within a given heap using only a few instructions. The system replaces object storage by manipulating page maps. Replacement transparency is satisfied since virtual addresses do not change.

Before or after the invocation of a parallel region, these logical heaps behave as *normal* program memory and support any form of access. During an invocation, Privateer changes the process' virtual page map, thus replacing the heaps' physical pages. This allows the runtime to replicate the storage for *all* objects in a heap, marking them with the copy-on-write page protection. Initially, values within the private heap appear identical to those from the sequential region. However, the OS traps updates to the private heap and silently duplicates those pages, thus isolating each worker's updates. The reduction heap is replaced and bytes within those pages are initialized with the identity value for the reduction operator.

Privateer validates most speculative properties with *instantaneous* checks—they can be determined at a point in the code and do not rely on history of previous operations. The speculative mapping of pointers to a particular heap can be checked by examining only the pointer address. The speculative restriction on the lifetime of short-lived objects can be checked at the end of each iteration. These properties are strong enough to provide the enabling benefits of speculation, yet induce only minimal runtime overhead.

Privacy validation is more complicated, requiring that we consider all operations that access a particular private object. A challenging aspect of privacy validation is allowing reads of values live-in to the loop. A worker who reads a live-in value must guarantee that no worker defined that value in an earlier iteration, requiring a flow of information among workers. The Privateer runtime system employs a two-phase approach to reduce the communication overhead of validation. The first phase occurs immediately, and detects several cases of privacy violation without any communication among the workers. The second phase completes the validation check by handling the cases of privacy violation that require communication. The second phase occurs during a checkpoint operation (see Section 5.2).

Upon entering the parallel region, the runtime also creates a *shadow* heap for each worker which has the same size as the private heap. Each byte of data within the private heap corresponds to a byte of metadata in the shadow heap. Privateer records metadata in the shadow heap about the history of accesses to private memory. This shadow heap is analogous to the shadow arrays in the LRPD technique [22]. Each byte of metadata contains a code indicating the history of that private byte given all information available to a worker. In particular, metadata contains enough information to determine whether a byte of private memory *may* contain a live-in value, or if it was necessarily defined during an earlier iteration of the parallel region. The interpretation of these codes is discussed in Section 5.1.

Since the compiler relies on profile driven speculative parallelization, the runtime system must support rollback and recovery in case of misspeculation. Privateer provides this via checkpointing. Speculative state is collected from all workers at regular intervals and validated for misspeculation. If no violations occur, then the checkpoint is marked non-speculative and used as a recovery point. Checkpoints are only collected and validated after a large number of iterations. This policy reduces checkpointing and validation overheads in the common case, but discards and recomputes a larger amount of work upon misspeculation.

4. The Privateer Analysis and Transformation

The Privateer system provides fully automatic analyses and transformations to privatize the data structures used by general purpose C and C++ applications. Figure 3 describes the compiler component. Each step is described in the following sections.

4.1 Profiling

The Privateer system uses a novel pointer-to-object profiler to connect dynamic pointer addresses with a set of object names. The profiler assigns static names to the memory objects of global or constant variables. The profiler names dynamic objects (e.g. `malloc` or `new`) or stack slots according to the instruction which allocates them and a *dynamic context*. The dynamic context distinguishes dynamic instances of a static instruction by listing the function and loop invocations which enclose that instruction.

The pointer-to-object profiler instruments the program to maintain a interval map from ranges of memory addresses to the name of the memory object which occupies that space, like [34]. This interval map enables the profiled program to determine the name of the object referenced by any pointer during a profiling run. The profiler instruments every pointer that cannot be mapped to a unique object at compile time. The profiler accumulates this information over program execution.

Finally, this profiler tracks the allocation and deallocation of memory objects with respect to dynamic contexts. This information allows the compiler to characterize the *lifetime* of objects and

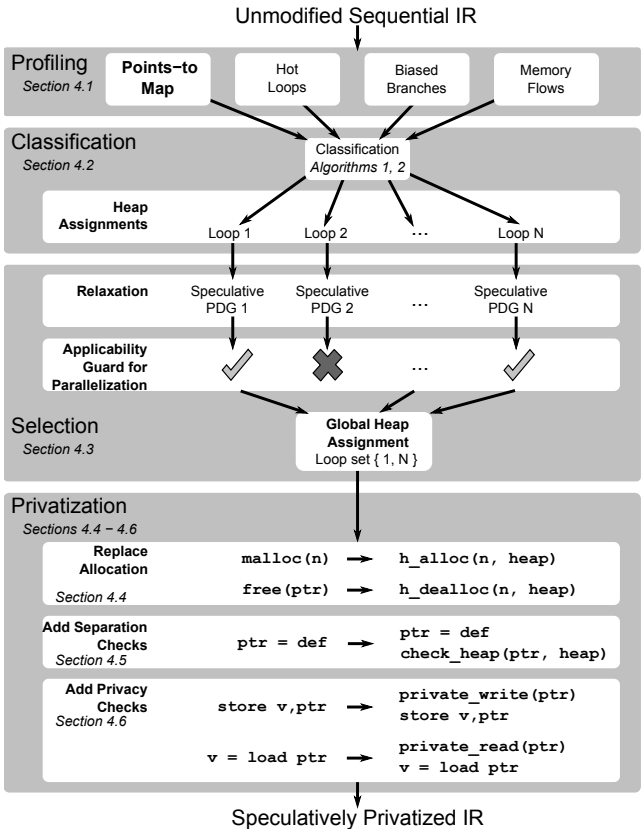

Figure 3: Structure of the Privateer Analysis and Transformation.

distinguish between short- and long-lived objects, supporting object lifetime speculation [13]. Short-lived objects exist only within a single iteration of a loop.

In Figure 2, the pointer-to-object map contains the following data. The pointer `qKill` on Line 27 always points to objects allocated by Line 11 in one of two contexts: either `enqueueQ` called at Line 60 or `enqueueQ` called at Line 74. All objects allocated on Line 11 are short-lived with respect to the loop on Line 46.

Privateer uses other profilers. A trip count profiler [15] identifies biased branches for control speculation (à la [5]). A memory flow dependence profiler similar to [4] augments static analysis. A value-prediction profiler guides value prediction speculation (à la [11]). Finally, an execution time profiler, similar to gprof [26], finds hot loops.

4.2 Classification

The hot loops access some objects in a restricted fashion. Using profile information, the system classifies each object as one of five access patterns: private, reduction, short-lived, read-only, and unrestricted. These labels are summarized in a *heap assignment*, which describes overall memory usage by mapping each object to one of the five heaps with restricted semantics.

Algorithm 1 determines a heap assignment for each loop. First, it calls `getFootprint` (Algorithm 2) to determine the read, write, and reduction footprints of the loop. These footprints are represented as sets of memory object names and may overlap. This function accumulates the objects written by a store operation or read by a load operation. The algorithm also identifies operation sequences which syntactically resemble an associative and commutative reduction operation. Limited profile *coverage* has minimal effect on Privateer's analyses, since such code is likely removed via control speculation.

Algorithm 1: *classify(L)*

```
let ShortLived = ∅; let Redux = ∅; let Unrestricted = ∅;
let Private = ∅; let ReadOnly = ∅;
let (ReadFootprint, WriteFootprint, ReduxFootprint) = getFootprint(L) ;
foreach object o ∈ WriteFootprint ∪ ReadFootprint do
    if Profile.isShortLived(o, L) then
        ShortLived = ShortLived ∪ { o } ;
    end
end
foreach object o ∈ ReduxFootprint do
    if (o ∉ ReadFootprint) and (o ∉ WriteFootprint) then
        Redux = Redux ∪ { o } ;
    end
end
let D = All cross-iteration memory flow dependences in L
      (assuming control and memory flow profiles)
foreach dependence (a → b) ∈ D do
    let (R_a, W_a, X_a) = getFootprint(a) ;
    let (R_b, W_b, X_b) = getFootprint(b) ;
    let F = (W_a ∪ X_a) ∩ (R_b ∪ X_b) ;
    Unrestricted = Unrestricted ∪ (F \ ShortLived \ Redux) ;
end
Private = WriteFootprint \ ShortLived \ Unrestricted \ Redux;
ReadOnly = ReadFootprint \ ShortLived \ Unrestricted \ Redux \ Private;
return (ShortLived, Redux, Unrestricted, Private, ReadOnly);
```

In Figure 2a, Privateer computes the footprint of the hot loop (Line 46), as follows. The read set contains the global queue structure `Q`, the global arrays `pathcost` and `adj`, and all linked list nodes allocated by Line 11. The write set contains `Q`, `pathcost`, and all linked list nodes. The reduction set is empty.

Algorithm 2: *getFootprint(S)*

```
let ReadFootprint = ∅; let WriteFootprint = ∅; let ReduxFootprint = ∅;
foreach instruction I in S do
    if I is of the form "r := load p" then
        let O = Profile.mapPointerToObjects(p);
        if (exists instruction of the form "store v, p") and
          (exists instruction of the form "v := op r, x" where
          op is associative and commutative) then
            ReduxFootprint = ReduxFootprint ∪ O;
        else
            ReadFootprint = ReadFootprint ∪ O;
        end
    end
    if I is of the form "store v, p" then
        let O = Profile.mapPointerToObjects(p);
        if (exists instruction of the form "r := load p") and
          (exists instruction of the form "v := op r, x" where
          op is associative and commutative) then
            ReduxFootprint = ReduxFootprint ∪ O;
        else
            WriteFootprint = WriteFootprint ∪ O;
        end
    end
    if I is of the form "r := call f(...)" then
        recur on f;
    end
end
return (ReadFootprint, WriteFootprint, ReduxFootprint)
```

The classification algorithm (Algorithm 1) partitions the loop's memory footprint across the five heaps according to access patterns. If an object is allocated and freed within an iteration, classification assigns it to the short-lived heap. If the compiler does not expect an object in the reduction set to be accessed by loads or stores elsewhere in the loop, classification assigns it to the reduction heap. This indicates that the reduction criterion is expected to succeed, but will still be verified at runtime via separation checks (Section 5.1). The unrestricted heap contains objects which partake in a loop-carried dependence, unless those objects were already assigned to the short-lived or reduction heaps. The private

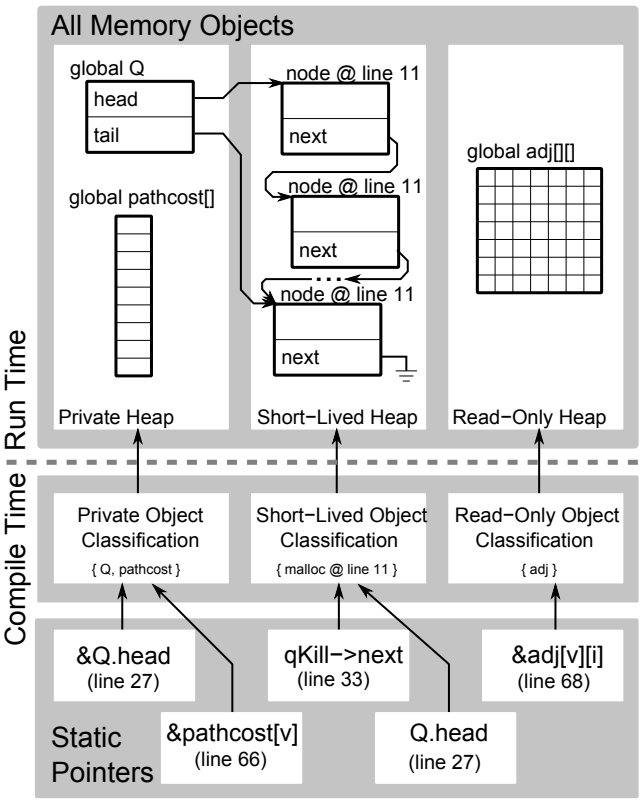

Figure 4: A heap assignment for Figure 2. Privateer speculatively separates objects into several classifications. Objects are allocated from *logical* heaps for efficient validation.

heap receives all other written objects. The read-only heap receives all other read objects. These five sets are collectively referred to as a *heap assignment*.

Figure 4 shows a heap assignment for the code in Figure 2a. The short-lived set contains all linked list nodes allocated at Line 11. The reduction and unrestricted sets are empty (and not shown). The private set contains the global queue structure `Q` and the global array `pathcost`. The read-only set contains the global array `adj`.

4.3 Selection

The compiler selects a subset of loops to parallelize from the set of hot loops with heap assignments. Program dependences are computed using static analysis and then refined according to the heap assignment:

- *The logical heaps are separated:* For a pair of operations o and p whose footprints are assigned to sets of heaps h_o and h_p respectively, if $h_o \cap h_p = \emptyset$ then remove all memory dependences $o \rightarrow p$ and $p \rightarrow o$.
- *The private, short-lived and reduction heaps eliminate loop-carried dependences:* For an operation o, if the footprint of o is contained in the private, short-lived, and/or reduction heaps, then for all operations x in the loop, remove all *loop-carried* memory dependences $o \rightarrow x$ and $x \rightarrow o$.

Additionally, dependences are refined with standard rules for value prediction, control speculation, and I/O deferral. The result is an optimistic view of program behavior in the expected case. This dependence structure is passed to parallelizing transformations to exclude inapplicable loops. The compiler selects the largest (by execution time) set of parallelizable loops subject to the following compatibility constraints. The compiler avoids nested parallelism: if the compiler finds (via static analysis or profiling results) that two loops may ever be simultaneously active, it marks the two loops incompatible. Second, two loops are incompatible if an object is assigned to different heaps for each loop. If two loops are incompatible, the compiler selects at most one of them. This selection process yields a single heap assignment for the set of selected hot loops.

4.4 Replace Allocation

Privateer replaces the allocation site for each object from the heap assignment. Storage for global objects is allocated from the appropriate heap during an initializer which runs before `main` (Lines 40–44), and is saved in a global variable (Lines 5–7). All uses of the addresses of such objects are replaced with loads from said global pointer. For stack allocations, the operation is replaced with an allocation from the appropriate heap and a corresponding deallocation is inserted at all function exits. Similarly, heap allocations and deallocations are replaced with the routines for the appropriate heap.

4.5 Add Separation Checks

The compiler inserts calls to trigger validation. To validate that a pointer refers only to objects within the correct heap, the compiler finds every static use of a pointer within the parallel region and traces back to the static definition of that pointer. It inserts calls to the `check_heap` function, which performs a separation check (see Section 5.1). Figure 2b shows a separation check on Line 29; other checks are proved successful at compile time and are elided.

4.6 Add Privacy Checks

To validate that private objects never partake in loop-carried flow dependences, the compiler finds every operation within the parallel region which accesses an object in the private heap. It inserts a call to `private_read` before loads, and a call to `private_write` before stores. These calls report address and access-size information to the runtime system, causing the runtime to validate privacy (see Section 5.1). In Figure 2b, Lines 15, 19 and 65 show privacy checks inserted by Privateer.

5. The Privateer Runtime Support System

The runtime support library serves several purposes. It manages the logical heaps and validates their speculative separation. It provides validation of speculative privacy. It coordinates periodic checkpoints and initiates recovery after a misspeculation. Parallel execution is governed by the parallelizing transformation applied to the program after privatization. In this investigation, the parallelizing transformation is DOALL.

Figure 5 shows a schematic time line of parallel execution with three workers. Speculative privatization does not add explicit communication between the workers, but requires periodic checkpoints, marked as CHKn. Additionally, each worker performs small inline misspeculation checks, denoted by grey bars. The figure shows a misspeculation at iteration $2k + 4$, followed by sequential, non-speculative recovery. Parallel execution resumes after recovery.

5.1 Runtime Validation of Speculation

Separate Heaps: The runtime system makes heavy use of the POSIX shared memory (shm) and memory map (mmap) facilities to achieve the desired separation model. Since workers must update their virtual memory maps independently, the Privateer runtime system uses processes and not threads. Heaps are created via `shm_open`. Each process maps them into its address space via `mmap` with read-only, read-write or copy-on-write protections.

The `mmap` facility allows the system to select a fixed, absolute virtual address for these heaps. Privateer exploits this feature by

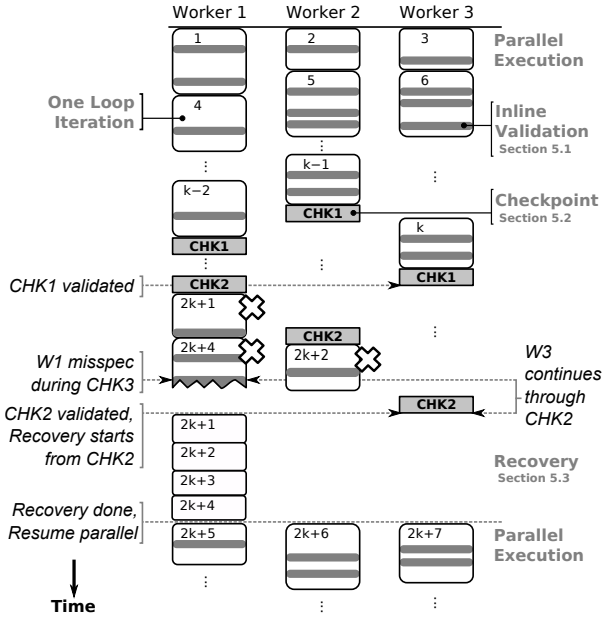

Figure 5: Example showing the worker processes during a parallel region. Iteration $2k + 4$ misspeculates, triggering a recovery.

hiding a *heap tag* within the heaps' virtual addresses. Bits 44–46 of the address hold a 3-bit heap tag, allowing the runtime to quickly determine if a pointer references an address within the correct heap. As a heap is subdivided by allocations, all objects within that heap inherit its tag. This choice of bit location was selected for compatibility with common operating systems and hardware, and allows 16 terabytes of allocation within any heap.

The privatizing transformation inserts a heap check at each instruction which computes a pointer address in the parallel region. This check indicates an assumed target heap for that pointer. The runtime tests the pointer's heap tag via bit arithmetic, reporting misspeculation upon mismatch. The bit patterns for the private and shadow heaps are chosen so they differ by only one bit. For a byte at address p within the private heap, the system computes the address of the corresponding byte of metadata in the shadow heap with a single bit-wise OR instruction.

Validating Short-Lived Objects: Each worker counts the number of objects allocated and not freed from its short-lived heap. If any of these objects is live at the end of an iteration, then lifetime speculation is violated, and the worker reports misspeculation [13].

Validating Privacy: Privacy is validated in two phases. First, a worker employs a fast test upon each access to private memory. This test requires no communication with other workers, but may fail to catch some violations. A thorough check will catch remaining violations during the checkpoint operation (see Section 5.2).

Every byte of metadata contains one of four codes: *live-in* (0), *old-write* (1), *read-live-in* (2), or a timestamp $3 + (i - i_0)$ encoding the iteration i after the most recent checkpoint i_0. Initially, the shadow heap contains all zero values (live-in). Privacy checks cause the runtime system to update metadata upon every private access. The transition rules for metadata are shown in Table 2. The simplest cases are the most common: a write to private memory updates the corresponding bytes of metadata with the current iteration timestamp; a read from private memory checks that the corresponding bytes of metadata match the current iteration timestamp. If the program ever reads a value that was defined by an earlier iteration, this can be detected by the fourth rule.

To support reading live-in values, the runtime marks a live-in byte with the code read-live-in. This indicates that a byte has been read, and appears to be a live-in value, but that privacy cannot be guaranteed without communicating with other workers. Instead, this property will be checked at the next checkpoint. If such a byte is overwritten before the checkpoint occurs, the system will conservatively report a misspeculation. Such a misspeculation may represent a false-positive. We selected this design since tests without false positives require a separate read-iteration timestamp, doubling the size of metadata. We did not observe false positives in practice.

Op.	Metadata		Comment
	Before	After	
Read	0	2	Read a live-in value.
	1	misspec	Loop-carried flow dependence.
	2	2	Read a live-in value.
	α ($2 < \alpha < \beta$)	misspec	Loop-carried flow dependence.
	β	β	Intra-iteration (private) flow.
Write	0	β	Overwrite a live-in value.
	1	β	Overwrite an old write.
	2	misspec	Conservative false positive.
	α ($2 < \alpha \leq \beta$)	β	Overwrite a recent write.

Table 2: Metadata transitions on private accesses. β is the timestamp for the current iteration, and α is the timestamp for an earlier iteration.

These metadata codes will eventually overflow a byte. A checkpoint resets the metadata range by replacing all writes before the checkpoint (metadata $\alpha \geq 3$) with old-write (1). Privateer triggers a checkpoint operation at least every 253 iterations.

5.2 Checkpoints

To support recovery, the speculative program periodically saves valid program state. The runtime selects a checkpoint period k before the parallel invocation. After every k-th iteration, worker processes copy their speculative state (the private, shadow, and reduction heaps) into a checkpoint object, as in Figure 5. This object is allocated by the first worker to reach that iteration and retired after the last worker reaches the iteration. The checkpoint system maintains an ordered list of checkpoint objects, each representing a distinct point in time, and allows arbitrarily many checkpoint objects. Workers acquire a lock on a single checkpoint object, not the whole checkpoint system, to avoid barrier penalties. This allows a fast worker to proceed to subsequent work units without waiting for slow worker processes to reach the checkpoint.

As mentioned in Section 5.1, privacy is validated by a two-phase approach. The runtime performs the *second* phase of validation as each worker adds its speculative state to the checkpoint object, using the same metadata transition rules as listed in Table 2. If misspeculation is detected while a worker is performing a checkpoint, that worker signals a misspeculation and aborts. Otherwise, that checkpoint object is marked non-speculative as soon as all workers have added their state to the checkpoint.

5.3 Recovery

If a worker detects misspeculation, it sets a global misspeculation flag and records the misspeculated iteration number. This worker terminates immediately, squashing all its speculative state created since its last checkpoint.

Since workers run at different speeds, it is possible that a remaining worker has not yet reached the checkpoint during which misspeculation occurred. Workers consult the global misspeculation flag after each iteration. If set, each worker compares its checkpoint ID $\lfloor i/k \rfloor$ against the ID of the checkpoint which misspeculated. If a worker has not yet reached the point of misspeculation, it continues execution; otherwise it terminates. This policy reduces wasted work upon misspeculation, as in Figure 5. If workers dis-

cover an *earlier* misspeculation before they terminate, they update the earliest iteration at which misspeculation occurs, and abort.

Once all worker processes have terminated, the main process begins non-speculative recovery. Using several calls to `mmap`, the main process replaces its heaps with those from the last valid checkpoint. The main process re-executes iterations non-speculatively until it has passed the iteration at which the earliest misspeculation occurred. Unless the program exits the loop during recovery, parallel execution resumes.

6. Evaluation

Privateer is evaluated on a shared-memory machine with four 6-core Intel Xeon® X7460 processors (24 cores total) running at 2.66 GHz with 24 GB of memory. Its operating system is 64-bit Ubuntu 9.10. The compiler is built on LLVM [15] revision 139148.

Privateer is evaluated with 5 programs that require speculative privatization for parallelization, as described in Table 3. Programs are selected from a set of C and C++ applications because their parallelization is limited by false dependences. We exclude many programs because they are parallelizable without Privateer. Some other programs feature data structures that Privateer can successfully privatize, but whose loops cannot be parallelized with DOALL because of real loop-carried flow dependences. We exclude those as well, since they are limited by DOALL, not by Privateer. More powerful parallelizing transformations, such as PS-DSWP [20] will be investigated in future work.

Specifically, we exclude `177.mesa` and `462.libquantum` since they can be parallelized without the aid of speculation, and thus we do not take credit for their performance. We exclude `164.gzip`, `256.bzip2`, and `456.hmmer` since the compiler cannot identify DOALL loops after Privateer's speculation has been applied. The compiler does not transform these codes.

Each benchmark is profiled with a training input (*train*). Performance evaluations are measured with a different testing input (*ref*). When we profile these with a third input (*alt*), the compiler generates identical code, suggesting that Privateer's analysis is reasonably stable with respect to profile input.

6.1 Parallel Performance Results

Figure 6 presents performance results generated by the fully automatic privatization and parallelization transform. These measurements are *whole application speedups* relative to the best sequential performance of the original application. The sequential applications are compiled with *clang -O3*.

These results indicate that privatization of data structures unlocks parallelization opportunities in these programs. Additionally, they indicate that Privateer's speculative separation is sufficiently powerful to reason about and operate on the dynamically allocated and irregular data structures present in these applications.

The `dijkstra` application from MiBench [12] reuses several data structures. It maintains a table of shortest paths and linked list of nodes whose shortest paths have changed—both as global variables. Successive iterations of the hot loop are synchronized by false dependences on these data structures. Privateer uses value prediction to speculate that the linked list is empty at the beginning of each iteration and privatizes the head node of the linked list and the shortest path table. The nodes within the linked list are assigned to the short-lived heap. Additionally, the hot loop includes calls to `printf` that are deferred into the speculative system, so that they may issue in any order yet commit in-order.

Privateer transforms the sequential version of the `swaptions` program from PARSEC [2]. It parallelizes the hot loop in the function `worker` by privatizing 17 memory objects, 15 of which are short-lived. The short-lived objects include a large number of vectors and matrices (arrays of pointers to row vectors) which

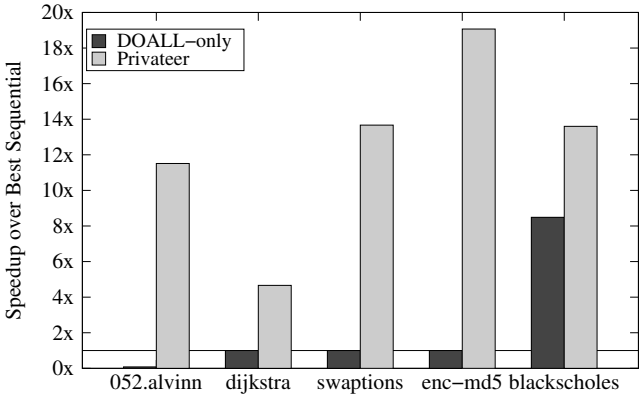

Figure 7: Enabling effect of Privateer at 24 worker processes.

are dynamically allocated at various points within `worker` and its callees, and passed around indirectly through other data structures. The LRPD-family techniques are inapplicable to this benchmark because of the linked matrix data structures.

The `052.alvinn` program is from SPEC [25]. To enable parallelization, Privateer privatizes four stack-allocated arrays. `052.alvinn` iterates over these arrays using pointer arithmetic and passes array references to callees, making static analysis difficult. Additionally, Privateer handles reductions on two global arrays and as well as a scalar local variable. At 8 cores, Privateer achieves a speedup of 5.66× on commodity hardware. OpenImpact [35] reports 6.44× with the help specialized hardware extensions. This compares favorably to STMLite+LLVM [17], which reports less than 2× in a software-only system with 8 cores.

The `enc-md5` program from Trimaran [28] computes message digests for a large number of data sets and prints each to standard output. Two factors limit parallelization of the programs outer loop: false dependences on the MD5 state object and digest buffer, and calls to `printf`. Privateer privatizes the state object and marks the digest buffer as short-lived. The side effects of stream output functions are issued through the checkpoint system and take effect only when the checkpoint is marked non-speculative.

Privateer transforms the sequential version of `blackscholes` from PARSEC [2]. In the hot loop-nest of this program, the inner loop is embarrassingly parallel. However, the outer loop cannot be parallelized directly because of output dependences on the pricing array, which is allocated in a different function. Privateer privatizes this array, allowing for parallel execution of the outer loop.

Figure 7 compares the performance of the DOALL transformation using 24 workers, with and without Privateer. "DOALL-only" refers to a non-speculative implementation which distributes loop iterations across worker threads, and thus does not incur checkpoint or validation overheads. Privateer enables parallelization of hotter loops. For `052.alvinn`, DOALL-only transforms a deeply nested inner loop. Performance gains do not outweigh the overhead of dispatching worker threads, and thus DOALL-only experiences slowdown. DOALL-only does not parallelize any loops in `dijkstra` or `enc-md5` because of real, frequent false dependences. The hot loop in `swaptions` is parallelizable but could not be proved parallelizable by our static analysis. DOALL-only parallelizes a hot inner loop in `blackscholes`; however, privatization allows the compiler to parallelize a hotter loop. Privatization enables the compiler to parallelize a single invocation, thus reducing spawn/join costs.

6.2 Overhead of the Runtime System

Privateer minimizes validation's runtime overhead. Figure 8 presents a breakdown of measured overheads for each program when using

Program	Dynamic				Replaced Static Allocation Sites					Extras
	Invoc	Checkpt	Priv R	Priv W	Private	Short-Lived	Read-Only	Redux	Unrestricted	
052.alvinn	200	2,600	8.2 GB	300 MB	4	0	4	3	0	-
dijkstra	1	5	84.9 GB	56.7 GB	10	3	11	0	0	Value, Control, I/O
blackscholes	1	5	0 B	4.0 GB	1	0	9	0	0	Value
swaptions	1	17	288 KB	169 KB	2	15	5	0	0	Value, Control
enc-md5	1	5	25.5 GB	30.8 GB	2	1	4	0	0	Control, I/O

Table 3: Details of privatized and parallelized programs, including number of invocations of the parallel region; total number of checkpoints constructed; total private bytes read and written; static number of objects assigned to each heap; and additional necessary transformation including value prediction speculation, control speculation, and deferral of I/O operations.

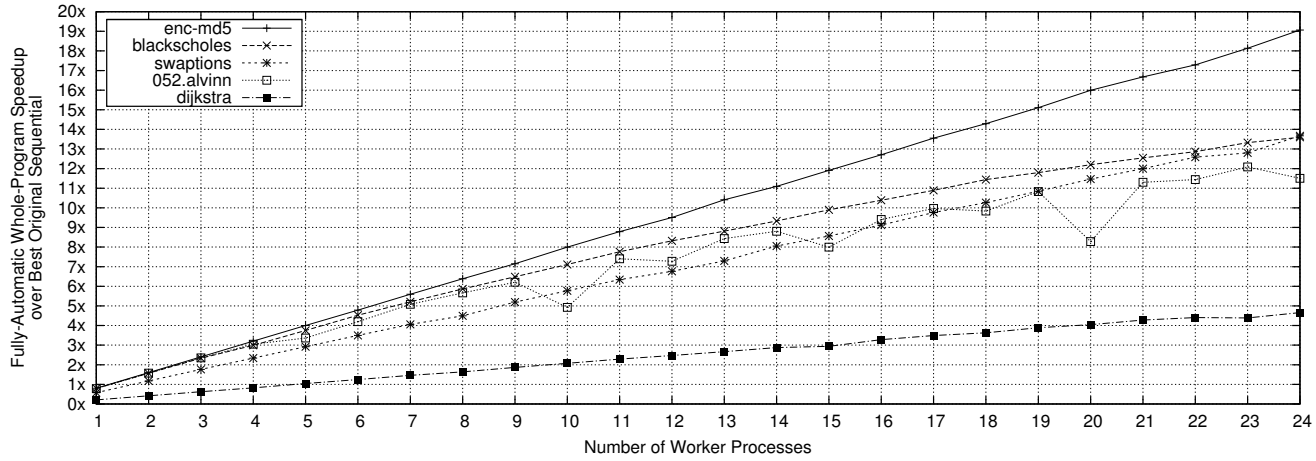

Figure 6: Whole program speedups of the fully automatically parallelized code, measured with respect to the best running time of the unmodified sequential application compiled with `clang -O3`. Each point is the average of three trials.

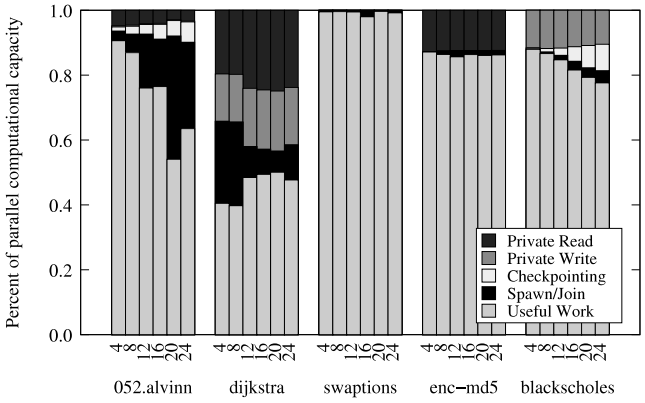

Figure 8: Breakdown of overheads on parallel performance.

4, 8, 12, 16, 20, and 24 worker processes. These numbers are normalized to the total computational capacity (CPU-seconds) of the parallel region: the number of processor cores times the duration of the parallel invocation. In these units, perfect utilization would be represented as 100% useful work. Overheads experienced during the parallel region subtract from utilization and prevent linear speedup. All times measure wall-clock time, not processor time: they include time spent blocking and context switching. If the parallel region invokes more than once, these numbers are the sum over all parallel invocations.

In the overheads figure, "Useful Work" refers to the portion of computational capacity spent executing instructions from the original sequential application. "Private Read" refers to the capacity spent updating metadata in response to a read from a private object. Similarly, "Private Write" refers to the bookkeeping for a write to a private object. "Checkpoint" refers to the capacity spent collecting, validating, and combining checkpoints.

Spawn refers to the unused capacity after a parallel invocation has begun, yet before the worker processes begin execution. This overhead is mostly determined by the latency of the operating system's implementation of `fork`. Join refers to the non-useful capacity after a worker process has finished its work units, yet before the parallel invocation has finished. This overhead is caused by four factors: imbalance among the workers, the latency of the worker-completed signal, the cost of installing the final non-committed state into the main process, and the cost of committing output operations that were issued during the parallel region. These two measurements are presented together as "Spawn/Join."

Results show that parallelized applications utilize most of the parallel resources for useful work. Both `052.alvinn` and `dijkstra` waste a significant amount of time joining their workers. This is caused by an imbalance in the latency of each worker, and a load balancing technique such as work stealing could potentially address this inefficiency. Validation of privacy is the next largest source of overhead. Percent of computational capacity used for privacy validation remained mostly constant as the number of workers increased, suggesting that the absolute amount of work for privacy validation grows with the number of workers.

6.3 Misspeculation Analysis

Privateer employs speculation to eliminate rare dependences and thus optimizes for the common case. To reduce the risk of misspeculation, Privateer interprets profiling results conservatively. No programs experienced misspeculation during evaluation. To better understand the effect of misspeculation, we inject artificial misspecu-

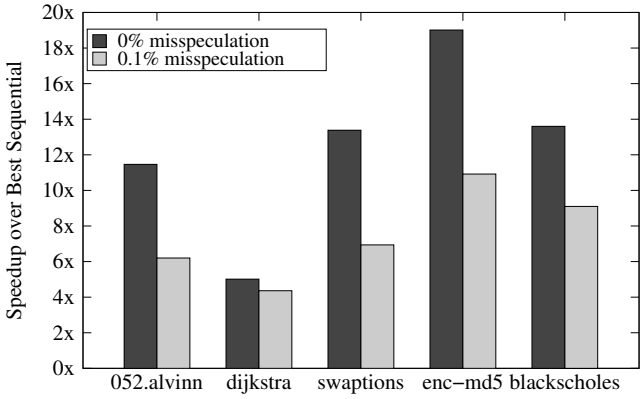

Figure 9: Performance degradation with misspeculation.

lation into the running application at fixed frequencies. The results of this experiment are shown in Figure 9. We present misspeculation rates as the percentage of *iterations* which misspeculate as opposed to *checkpoints*, since iterations are more standard. Privateer's recovery mechanism operates at the granularity of checkpoints (see Section 5.2). Thus, a misspeculation rate of 0.1% causes about one in four checkpoints to fail. For `blackscholes`, we increased the input size so that the hot loop executed at least 1,000 iterations.

For most programs, these results indicate that Privateer's performance benefits are sensitive to misspeculation. Four of five programs lose half of their speedup with a misspeculation rate of 0.1%. This suggests that Privateer requires high-confidence speculation for performance.

7. Related Work

Paralax [32] uses privatization to enable parallelization. The authors note that privatization analysis is difficult on C programs. They propose KILL annotations to assert the absence of flow dependences through a data structure, indirectly answering the privatization criterion. These annotations are applied to named objects or object referenced by a single pointer indirection. This prevents the application of KILL to recursive data structures.

Early works on privatization [16, 29] are limited by the strength of static analysis on the privatization criterion and memory layout problems. The PD Test [21] reduces reliance on static analysis by adding inspector loops to dynamically verify the privatization criterion at runtime. Similarly, Hybrid Analysis [24] uses a generalized representation for indirect array references to statically generate predicates, which are then resolved at runtime for dynamic privatization. The LRPD [22] and R-LRPD [7] Tests obviated the need for static analysis by evaluating the privatization criterion speculatively. All of these techniques are evaluated on array-based codes written in FORTRAN and cannot handle pointers, linked lists, and other dynamic data structures.

Array Static Single Assignment (ASSA) [14] extends Static Single-Assignment form [6] to arrays. ASSA requires that any named memory location has exactly one definition. Repeated updates are represented with new static names and joined via ϕ-nodes. In this form, false dependences do not exist, and a compiler may distribute operations across threads considering only flow dependences. However, pointer indirection allows for ambiguous updates and foils ASSA analysis. Array Expansion [10] and Dynamic Single Assignment (DSA) [31] are similar to ASSA. Instead of creating new names, these add a new dimension to arrays representing the new definition. Instead of inserting ϕ-nodes, DSA emits instructions to explicitly select the appropriate value at control join points. Region Array SSA [23] uses partial aggregation of array regions to reduce the runtime overhead of ASSA. These provide the same single-assignment semantics as ASSA, and suffer from the same applicability problems in light of unrestricted pointers and casts. A representative DSA [31] is inapplicable to loops which contain loads or stores from pointers.

Software Transactional Memory (STM) systems [8, 17, 18] provide isolation and consequently privatize data structures written during a transaction. To detect conflicts, these techniques keep a log of memory accesses for offline validation. STMLite integrates an automatic DOALL compiler featuring several enabling transformations [17] and implemented in LLVM [15]. STMLite's central commit process can quickly become an execution bottleneck. The other transactional systems are not evaluated in an automatic system; weak static analysis may cause a large volume of unnecessary validations, and it is unclear whether these systems scale to that volume. None of these STMs provide speculative reduction support, and so a compiler must rely on a static criterion.

The CorD+Objects [27] compiler and STM reduce copy overheads by tracking speculative state of *objects*. To address replacement transparency, the compiler transforms pointers into "double pointers" and the runtime maintains a map between copies of an object. This transformation assumes that all accesses conform to the object's declared type, but may fail due to reinterpretation casts. Static analysis cannot always determine whether an object is ever reinterpreted. The transformation also assumes that all pointer values are visible in the IR, but C's weak types allow "disguised" pointers, as discussed in [3]. Like STMs, CorD+Objects does not support speculative reductions. Since Privateer provides replacement transparency using virtual page mapping, its compiler has no need to identify or manipulate pointer values in the IR.

Several works modify the default process memory model by manipulating virtual memory maps. DoublePlay [33] employs the copy-on-write mechanism to isolate different epochs of a single process, providing a deterministic replay facility. Grace [1] implements a safe multithreading programming model to reduce development effort for parallel programs. Behavior oriented parallelization [9] provides a speculative execution model that resembles an STM and features an optimized value-based misspeculation detection system. These works are intended as programmer tools to aid the development of parallel applications, yet none automatically parallelize applications.

8. Conclusion

Automatic parallelization is a promising strategy to deliver scalable application performance on parallel architectures. Privateer enables a compiler to extract more parallelism by selectively privatizing data structures. Privateer's heap separation enables greater applicability than related techniques, and allows for efficient validation. Privateer's fully automatic privatization and parallelization delivers a geomean whole-program speedup of 11.4× over best sequential execution for 5 programs on a 24-core shared memory machine.

Acknowledgments

We thank the entire Liberty Research Group for their support and feedback during this work. We also thank the anonymous reviewers for their insightful comments. Additionally, we thank Andrew Appel, Gordon Stewart, Lennart Beringer, Jude Nelson and Daya Bill for commenting on early drafts. This material is based on work supported by National Science Foundation Grant 0964328 and DARPA contract FA8750-10-2-0253. Prakash Prabhu thanks Google, Inc. for fellowship support. This work was carried out while Ayal Zaks was visiting Princeton University, supported by the HiPEAC network of excellence, and on leave from IBM Haifa Research Lab.

References

[1] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: safe multithreaded programming for C/C++. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, 2009.

[2] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[3] H.-J. Boehm. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming Language Design and Implementation*, pages 89–98, New York, NY, 1996. ACM.

[4] T. Chen, J. Lin, X. Dai, W.-C. Hsu, and P.-C. Yew. Data dependence profiling for speculative optimizations. In E. Duesterwald, editor, *Compiler Construction*, volume 2985 of *Lecture Notes in Computer Science*, pages 2733–2733. Springer Berlin / Heidelberg, 2004.

[5] W. Y. Chen, S. A. Mahlke, and W. W. Hwu. Tolerating first level memory access latency in high-performance systems. In *Proceedings of the 1992 International Conference on Parallel Processing*, pages 36–43, Boca Raton, Florida, 1992. CRC Press.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[7] F. H. Dang, H. Yu, and L. Rauchwerger. The R-LRPD test: Speculative parallelization of partially parallel loops. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, pages 20–29, 2002.

[8] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Distributed Computing*, pages 194–208, 2006.

[9] C. Ding, X. Shen, K. Kelsey, C. Tice, R. Huang, and C. Zhang. Software behavior oriented parallelization. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 223–234, New York, NY, 2007. ACM.

[10] P. Feautrier. Array expansion. In *Proceedings of the 2nd International Conference on Supercomputing*, pages 429–441. ACM, 1988.

[11] F. Gabbay and A. Mendelson. Can program profiling support value prediction? In *Proceedings of the 30th annual ACM/IEEE International Symposium on Microarchitecture*, pages 270–280, Washington, DC, 1997. IEEE Computer Society.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, 2001. IEEE Computer Society.

[13] H. Kim, N. P. Johnson, J. W. Lee, S. A. Mahlke, and D. I. August. Automatic speculative DOALL for clusters. *Proceedings of the 10th IEEE/ACM International Symposium on Code Generation and Optimization*, March 2012.

[14] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 107–120, 1998.

[15] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the Annual International Symposium on Code Generation and Optimization*, pages 75–86, 2004.

[16] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 2–15, New York, NY, 1993. ACM.

[17] M. Mehrara, J. Hao, P.-C. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[18] Y. Ni, A. Welc, A.-R. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. In *Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 195–212, 2008.

[19] C. G. Quiñones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 269–279, New York, NY, 2005. ACM.

[20] E. Raman, G. Ottoni, A. Raman, M. Bridges, and D. I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the Annual International Symposium on Code Generation and Optimization*, 2008.

[21] L. Rauchwerger and D. Padua. The Privatizing DOALL test: A run-time technique for DOALL loop identification and array privatization. In *Proceedings of the 8th International Conference on Supercomputing*, pages 33–43, New York, NY, 1994. ACM.

[22] L. Rauchwerger and D. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *ACM SIGPLAN Notices*, 30(6):218–232, 1995.

[23] S. Rus, G. He, C. Alias, and L. Rauchwerger. Region Array SSA. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 43–52. ACM, 2006.

[24] S. Rus, L. Rauchwerger, and J. Hoeflinger. Hybrid analysis: static & dynamic memory reference analysis. *International Journal of Parallel Programming*, 31:251–283, August 2003.

[25] Standard Performance Evaluation Corporation. http://spec.org.

[26] The GNU Project. GNU Binutils. http://gnu.org/software/binutils.

[27] C. Tian, M. Feng, and R. Gupta. Supporting Speculative Parallelization in the Presence of Dynamic Data Structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[28] Trimaran. Trimaran Benchmarks Packages. http://trimaran.org.

[29] P. Tu and D. A. Padua. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, pages 500–521, 1994.

[30] N. Vachharajani, R. Rangan, E. Raman, M. J. Bridges, G. Ottoni, and D. I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, pages 49–59, Washington, DC, 2007. IEEE Computer Society.

[31] P. Vanbroekhoven, G. Janssens, M. Bruynooghe, and F. Catthoor. A practical dynamic single assignment transformation. *ACM Transactions on Design Automation of Electronic Systems*, 12, September 2007.

[32] H. Vandierendonck, S. Rul, and K. De Bosschere. The Paralax infrastructure: Automatic parallelization with a helping hand. In *Proceedings of the 19th International Conference on Parallel Architecture and Compilation Techniques*.

[33] K. Veeraraghavan, D. Lee, B. Wester, J. Ouyang, P. M. Chen, J. Flinn, and S. Narayanasamy. Doubleplay: parallelizing sequential logging and replay. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 15–26, New York, NY, 2011. ACM.

[34] Q. Wu, A. Pyatakov, A. N. Spiridonov, E. Raman, D. W. Clark, and D. I. August. Exposing memory access regularities using object-relative memory profiling. In *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2004.

[35] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *Proceedings of the 14th International Symposium on High-Performance Computer Architecture*, 2008.