Automatic Detection of Large Extended Data-Race-Free Regions with Conflict Isolation

Alexandra Jimborean, Per Ekemark, Jonatan Waern, Stefanos Kaxiras, and Alberto Ros

Abstract—Data-race-free (DRF) parallel programming becomes a standard as newly adopted memory models of mainstream programming languages such as C++ or Java impose data-race-freedom as a requirement.
We propose compiler techniques that automatically delineate extended data-race-free (xDRF) regions, namely regions of code that provide the same guarantees as the synchronization-free regions (in the context of DRF codes). xDRF regions stretch across synchronization boundaries, function calls and loop back-edges and preserve the data-race-free semantics, thus increasing the optimization opportunities exposed to the compiler and to the underlying architecture. We further enlarge xDRF regions with a conflict isolation (CI) technique, delineating what we call xDRF-CI regions while preserving the same properties as xDRF regions. Our compiler (1) precisely analyzes the threads' memory accessing behavior and data sharing in shared-memory, general-purpose parallel applications, (2) isolates data-sharing and (3) marks the limits of xDRF-CI code regions. The contribution of this work consists in a simple but effective method to alleviate the drawbacks of the compiler's conservative nature in order to be competitive with (and even surpass) an expert in delineating xDRF regions manually.
We evaluate the potential of our technique by employing xDRF and xDRF-CI region classification in a state-of-the-art, dual-mode cache coherence protocol. We show that xDRF regions reduce the coherence bookkeeping and enable optimizations for performance (6.4%) and energy efficiency (12.2%) compared to a standard directory-based coherence protocol. Enhancing the xDRF analysis with the conflict isolation technique improves performance by 7.1% and energy efficiency by 15.9%.

Index Terms—Compile-time analysis, inter-procedural analysis, inter-thread analysis, data sharing, data races, cache coherence.

1 Introduction

Parallel programming languages based on the shared-memory model have well-defined memory consistency models to clarify when data modified by one thread must be visible to other threads. To simplify reasoning about correctness of parallel executions, mainstream languages such as C++ and Java have already adopted data-race-free (DRF) as a standard and provide none or weak guarantees in the presence of data races. For instance, C and C++ programs that contain data races have undefined semantics [1], [2], [3]. In contrast, data-race-free codes enable a variety of optimizations based on the fundamental observation that different threads cannot access the same memory location without synchronization, if at least one thread modifies the target variable.

In other words, in DRF applications, *synchronization-free regions* provide the strong guarantee that different threads cannot target concurrently the same memory address. Leveraging this property, recently proposed micro-architectural enhancements relax unnecessarily restrictive constraints, as shown for example in state-of-the-art coherence protocols [4], [5], [6], [7], [8], [9], [10]. These proposals demonstrate that synchronization-free regions in DRF applications permit the core to delay the action of publishing the writes, shown in Figure 1.b, leading to significant improvements in performance and energy compared to traditional protocols (Figure 1.a). Similarly, C/C++ compilers and alike typically optimize synchronization-free regions as if the code was sequential, without speculation or costly inter-thread analysis.

In this paper we denote synchronization-free regions that are

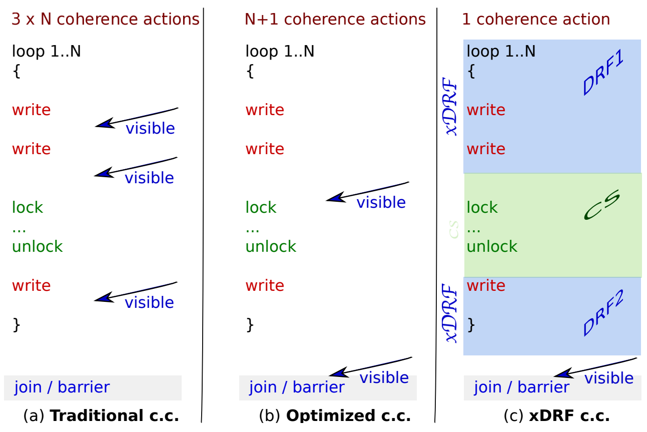


Fig. 1. (a) A standard cache coherence (c.c.) protocol makes the $write$ operations visible immediately after they have executed, thus performing 3×N actions. (b) Coherence protocols designed for DRF applications delay the action of making write operations visible until the first encountered synchronization point, hence $N+1$ actions. (c) The xDRF region consists of both DRF1 and DRF2 regions (bypassing CS). An xDRF-aware cache coherence protocol can safely defer the action of publishing writes until the boundary of the xDRF region, thus significantly reducing the number of actions to only one action.

not guarded by lock-unlock operations as DRF. *Extended data-race-free (xDRF)* regions are sets of DRF regions which span across synchronization points (e.g. acquire-release pairs), bypass the synchronized code (i.e. the critical section), while maintaining the DRF semantics [11] across the entire region [12], [13]. For example, in Figure 1.c, the xDRF region consists of the data-race-free regions *DRF1* and *DRF2*, excluding the synchronized code which we denote as *enclave* non-DRF region (*CS*).

In short, xDRF regions enable optimizations across synchro-

• *A. Jimborean, P. Ekemark, J. Waern, and S. Kaxiras are with the Department of Information Technology, Uppsala University, 751 05 Uppsala, Sweden. E-mail: alexandra.jimborean@it.uu.se, stefanos.kaxiras@it.uu.se*
• *A. Ros is with the Computer Engineering Department, University of Murcia, 30100 Murcia, Spain. E-mail: aros@ditec.um.es*

nization points. At the compiler level, xDRF regions enable thread-local (sequential) reasoning and static optimizations across the entire xDRF region, without the need for whole-code-analysis. Unlike standard optimizations, xDRF-aware optimizations can bypass synchronization points (pairs of acquire-release) and function calls. At the micro-architectural level, xDRF region classification translates to a private (thread-local) vs. shared classification of accesses, which is essential for efficient data placement, designing optimized coherence protocols or reordering memory operations. In this proposal we emphasize the benefits of xDRF regions with an xDRF-aware cache coherence protocol and leave compile-time optimizations and other micro-architectural optimizations for future work.

We have previously demonstrated that an xDRF-aware cache coherence protocol shows significant performance and energy improvements, but the scope of the xDRF analysis was previously restricted to automatically parallelized and OpenMP applications [12], [13]. This line of research demonstrated that *structured parallel programming* (OpenMP, TBB, Cilk, etc) provides strong guarantees, which can be exploited to delimit xDRF regions with high accuracy. To address a considerably larger class of applications, we describe compiler techniques to statically identify xDRF regions in "unmanaged" shared-memory parallel applications that follow the fork-join with synchronization model (e.g. based on pthreads) [14]. They represent the most challenging class of codes for static analysis due to the use of pointers, indirections, complex control-flow, recursions, etc. In contrast to previous work that relies on the programming paradigm (OpenMP) [12], [13], we target applications where the programmer, not the compiler, has control over the way parallelism is expressed.

Departing from DRF applications, we proposed a compile-time technique for identifying xDRF regions [14]. To verify the xDRF properties, the compiler checks whether memory accesses initiated from different data-race-free regions target the same location and must also analyze whether these accesses alias the memory locations accessed from critical sections. Next, the inter-thread, inter-procedural analysis identifies synchronization points that lock the *same synchronization variables* and cross-checks whether the memory accesses *before* the critical section of one thread share data (i.e. conflict) with the accesses *after* the matching critical section of another thread. If such a conflict is detected, the conservative xDRF analysis would mark the synchronization point as an xDRF boundary.

Due to pointer aliasing, more conflicts are detected at compile-time than actually occur in practice (false positive). Consequently, aliasing leads to a larger amount of xDRF limits than necessary [14]. Smaller xDRF regions reduce the opportunity for optimization, and therefore, lead to lower performance and energy efficiency than the actual potential of the applications.

This work performs an extensive analysis of xDRF regions and proposes a novel mechanism that isolates conflicts (aliases detected at compile time) in order to achieve larger xDRF regions, that is, less xDRF region boundaries. Large xDRF regions with conflict isolation xDRF-CI improves our previous xDRF analysis [14] through the following contributions:

1) We propose a technique to reduce the side-effects of the compiler's conservative nature and to identify xDRF regions with the same accuracy as (or even surpassing) an expert (section 3, subsubsection 4.5.2, subsubsection 5.2.2).
2) Insights regarding the advantages and disadvantages of large xDRF regions (section 3, section 6).

We evaluate the potential of the xDRF classification with a state-of-the-art, dual-mode cache coherence protocol [13] which deactivates coherence during the execution of xDRF regions and maintains coherence in hardware for the rest of the accesses. We report improvements in execution time (6.4%) and energy efficiency (12.2%) compared to a standard directory-based protocol. Furthermore, when we add the conflict isolation technique improvements go up to 7.1% in performance and 15.9% in energy efficiency. In *Barnes*, for example, execution time and energy consumption is reduced by 7.8% and 34.4%, respectively when compared to xDRF without conflict isolation, matching or even outperforming the manually annotated version.

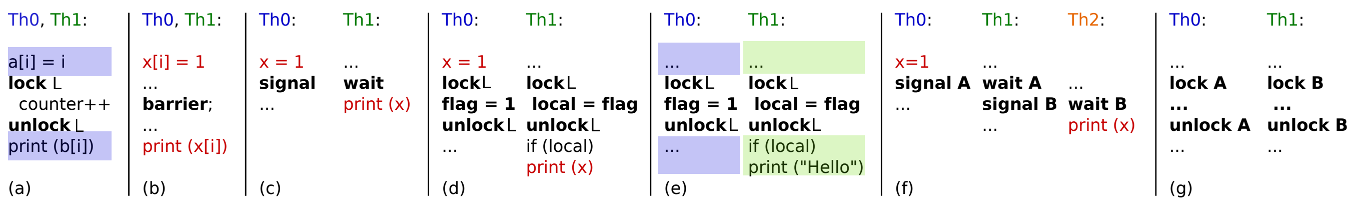
2 Related Work

We have previously proposed methods to identify and exploit xDRF regions in OpenMP applications [12], but those techniques are not suitable for "unmanaged" (e.g. pthreads) parallel applications based on the fork-join with synchronization model, addressed in this work.

Joisha et al [15] build a Procedural Concurrency Graph to determine interferences between threads and identify accesses with read- and write-siloed properties on certain intraprocedural paths. The analysis unblocks classical compiler optimizations for accesses free of interferences. Similarly, Effinger-Dean et al. [16], [17] perform a data-centric classification of regions, called interference free regions (IFR). IFRs are associated to variables (data), extend forward until the first `release` and backwards until the first `acquire` operation, and ensure that no other thread accesses the certain data during the IFR execution. Our compiler analysis ensures that *all* memory accesses within the xDRF region are free of interferences and can expand both backwards and forward across multiple acquire-release operations, across function boundaries and loop back-edges.

Techniques for private-shared data classification [18], [19], [20] consider memory blocks as shared if accessed by different threads at different execution points (i.e. in different regions). xDRF takes temporality into consideration and classifies such accesses as private throughout the xDRF region. Singh et al. [18] propose a static thread-escape analysis which identifies as "safe" (i.e. private) only data that is guaranteed to be thread-local or read-only, while dynamically allocated variables, global or static variables are marked as unsafe. Moreover, an instruction which can access both safe and unsafe data (e.g. a pointer dereference), would demote all safe data it may touch to unsafe. In consequence, safe data is restricted only to locations that are thread-local and can only be accessed by safe instructions.

A wide spectrum of static and dynamic techniques have been proposed [17], [21], [22], [23], [24], [25] to combat races. Static techniques [21], [22], [23] must be conservative and therefore report false-positives, while dynamic techniques [17], [24], [25], [26] miss races which do not occur in the observed execution and introduce high overheads. Acculock [27] is a hybrid lockset happens-before data race detector, balancing precision and coverage by exploring thread interleavings which do not occur in the observed execution. Valor [28] is a software-only, dynamic data race detector which operates at region level using epochs to identify ongoing regions and logs to keep track of read/write operations. Conflict Exceptions [26] relies on hardware support



```
(a)
Th0, Th1:

a[i] = i
lock L
  counter++
unlock L
print (b[i])

(b)
Th0, Th1:

x[i] = 1
...
barrier;
...
print (x[i])

(c)
Th0:        Th1:

x = 1       ...
signal      wait
...         print (x)

(d)
Th0:        Th1:

x = 1       ...
lock L      lock L
flag = 1    local = flag
unlock L    unlock L
...         if (local)
            print (x)

(e)
Th0:        Th1:

...         ...
lock L      lock L
flag = 1    local = flag
unlock L    unlock L
...         if (local)
            print ("Hello")

(f)
Th0:        Th1:        Th2:

x=1         ...
signal A    wait A      ...
...         signal B    wait B
            ...         print (x)

(g)
Th0:        Th1:

...         ...
lock A      lock B
...         ...
unlock A    unlock B
...         ...
```

Fig. 2. In (a) and (b) multiple threads execute the same code and cross a synchronization point, while (c) - (f) show examples when the threads execute different code regions that synchronize on the same resource. In (g) threads do not lock the same resource, hence the regions do not synchronize one with the other and do not contribute to the xDRF delimitation of the other thread. Conflicts are marked in a red square indicating that the synchronization point splits the code executed by each thread in two different xDRF regions. Shaded blocks indicate xDRF regions (one per thread). xDRF regions bypass and enclave synchronization points, but do not include them. (xDRF regions that cannot expand across synchronization points are not shaded –(b), (c), (d), (f).)

for race-detection. In contrast, our xDRF analysis is entirely static, therefore region classification is available prior to execution suitable for both compiler and micro-architectural optimizations.

Our work goes along the lines of data race detectors, but is *not* a data race detector. xDRF builds upon the premises that the code is DRF and identifies large regions of code which preserve the DRF semantics.

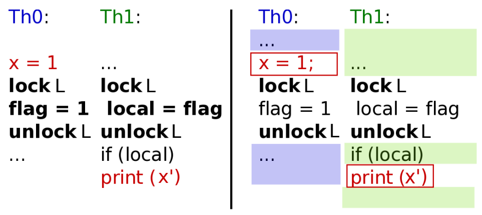
3 What Are xDRF and xDRF-CI Regions?

xDRF and xDRF-CI regions share the property that they preserve the DRF semantics across multiple synchronization points, but differ in the manner in which they mark synchronization points as enclave or as xDRF boundaries. This section first describes the xDRF regions, followed by the xDRF-CI regions.

We start with a few intuitive examples (Figure 2) illustrating cases when the xDRF bypasses and extends beyond the synchronization point –(a) and (e)– and other cases when conflicts between the regions preceding and following the synchronization point forces the split of the xDRF region executed by each thread –(b), (c), (d), (f). Synchronization operations shown in these examples use the same synchronization variable, unless indicated otherwise. We consider that the entire critical section represents a synchronization point (denoted non-DRF, on in short nDRF). Conflicts can occur only between accesses that escape the thread scope. The example in (a) shows an xDRF region that contains the memory accesses to $a[i]$ and $b[i]$ (assuming that within the same array, threads access different elements). Since $a[i]$ and $b[i]$ are different, threads Th_0 and Th_1 are free to reorder the memory accesses across the synchronization point because these accesses share no data. In (b) on the other hand, it may be that while Th_0 initializes $x[i]$, Th_1 prints its value, hence the barrier represents a limit between two different xDRF regions in which different threads may access the same element of an array. The first xDRF region contains the initialization $x[i] = 1$ and the second xDRF region prints $x[i]$. In (c) the threads share x, hence the signal-wait becomes a limit between consecutive xDRF regions. $xDRF_1$ of Th_0 contains $x = 1$ and $xDRF_2$ of Th_0 contains the region after synchronization illustrated as $\{...\}$ (similarly for Th_1). In (d) the signal-wait mechanism is implemented by means of flags, but the conflict to x, again, forces the split of the region in distinct xDRF regions. In contrast, in (e), there are no conflicts between the synchronization-free regions before and after the critical sections, therefore there is no need to split the region. Each thread considers the critical section as *enclave* (not breaking the xDRF region) in its xDRF region and can freely reorder memory accesses within the xDRF region across the synchronization point, as long as intra-thread dependences are respected (i.e. the accesses to the local variable $local$). The more complex example in (f) shows three threads that synchronize by transitivity. Since Th_0 and Th_1 synchronize using $signal\ A$ - $wait\ A$ and, similarly, Th_1 and Th_2 synchronize using $signal\ B$ - $wait\ B$ there is an implicit synchronization between Th_0 and Th_2, thus the accesses performed by all three threads have to be cross-checked for conflicts. Since Th_0 and Th_2 both access variable x and Th_0 performs a write, the synchronization points that separate the accesses to x – namely $signal\ A$ - $wait\ A$ and $signal\ B$ - $wait\ B$– are marked as *non-enclave* (breaking the xDRF regions), which means that accesses cannot be reordered across these synchronization points because they are shared between the threads. In consequence, (f) illustrates six xDRF regions, i.e. one before and one after each synchronization point. Finally, (g) shows that it is necessary to check for conflicts only between threads that synchronize on the same variable. Threads executing the regions in (g) do not synchronize one with another since they lock different resources.

The driving force of the xDRF analysis is that, in a DRF application, conflicts cannot occur between memory accesses executed outside critical sections. If two memory accesses that belong to data-race-free regions (DRF) executed by different threads target the same data (e.g. Figure 2 (c)), the synchronization point adjacent to the DRF regions imposes a *happens-before* relation and represents an xDRF boundary. Thus, the xDRF analysis merely verifies whether any memory access performed *before* the synchronization (i.e. before the lock) conflicts with a memory access performed by any other thread *after* a synchronization operation that uses the same resource (i.e. after the unlock). Furthermore, for completeness, the analysis must ensure that the critical sections do not target the same data as a DRF region. For example, in Figure 2 (d), the programmer may conservatively (or in error) place the instruction $if(local)print(x)$ performed by T_1, inside the critical section. Although no conflict occurs on the paths preceding and following the matching synchronizing operations, the region is not xDRF since the two threads share variable x and one access is outside a synchronizing operation.

Compilers are nevertheless limited in their ability to disambiguate memory operations and, in the presence of, for example, dynamic memory allocation, many false-positive conflicts are reported. To diminish the negative impact of the conservative compile-time decisions, we consider that only synchronization of the type barriers, joins and signal-waits represent xDRF boundaries, since these synchronization mechanisms impose an ordering of threads and imply sharing of data between threads, before and after the synchronization point. All other synchronization points are marked as enclave, allowing the xDRF region to extend beyond



```
Th0:        Th1:          Th0:        Th1:
                          ...
x = 1       ...           x = 1;      ...
lock L      lock L        lock L      lock L
flag = 1    local = flag  flag = 1    local = flag
unlock L    unlock L      unlock L    unlock L
...         if (local)    ...         if (local)
            print (x')                print (x')
```

Fig. 3. Comparison of the xDRF (left) and xDRF-CI (right) analysis and delineation of regions. In the xDRF analysis, a conflict (accesses to x, shown in red) turns the synchronization point into an xDRF boundary. In the xDRF-CI analysis, the conflict is isolated (exposed to the optimizations), enabling the xDRF-CI region to safely extend across the synchronization point. The reasoning behind this delineation decision is emphasized in the case when the two pointers x and x' may alias. The xDRF analysis breaks the xDRF region unnecessarily upon a false-positive conflict (MayAlias pointers rarely translate to true conflicts in practice [29]), while the xDRF-CI delineation simply isolates the conflict, without breaking the xDRF-CI region.

them. Next, to ensure correctness, potential conflicts (i.e. memory accesses targeting the same location) are isolated (i.e. marked as enclave nDRF regions) to be exposed to subsequent optimizations, as shown in Figure 3.

A natural extension would be to mark all synchronization points as enclave and to isolate all conflicts. However, this would extend the xDRF region to the entire program, increasing the probability that many pairs of memory accesses alias. As a result, most of the memory accesses would actually be isolated, leaving the xDRF region with few (or no) memory accesses to optimize. In contrast, the pairs of accesses within a smaller region do not alias, enabling optimizations within that region.

We aim to strike a balance between the size of the xDRF region and the potential for optimizations. On one hand, an xDRF region that spans a small code region would not contain sufficient memory accesses for the optimizations to have a significant impact on performance. On the other hand, increasing the size of the xDRF region to span a too large fraction of the program, would entail more aliasing memory operations and would force their exclusion from the xDRF region. Thus, the xDRF region, despite spanning a large part of the code, would contain very few memory accesses, reducing the potential for optimizations. Our solution breaks xDRF regions on barriers, joins, and signal-waits, which by their semantics entail that threads share data, but allows the xDRF region to extend across critical sections. This solution provides a good balance between the time data remains private to the thread (size of the xDRF region) and the optimization opportunity (number of memory operations contained by the xDRF region, i.e. that are not isolated).

We define *consecutive regions* to be regions of code reachable by control-flow without passing through other regions of the same type. For example, in Figure 1, $DRF1$ and $DRF2$ are consecutive DRF regions, since there is a path from $DRF1$ to $DRF2$ that does not cross any other DRF region, although it crosses a non-DRF (CS) region. Intuitively, an xDRF region consists of consecutive data-race-free regions executed by one thread, with the property that the accesses performed during the xDRF region do not target a memory location accessed by any other *concurrent* thread. We denote two non-DRF regions as *matching nDRF regions* if they synchronize using the same variable, (Figure 2 (a), (d), (e) $lockL$-$unlockL$ from Th0 matches the nDRF from Th1 aince they synchronize on the same variable L). And, by extension, xDRF regions corresponding to different threads are called *matching xDRF regions*, if they enclave matching nDRF regions (Figure 2 (e)). We guarantee that:

- Threads executing matching xDRF regions do not access the same memory location, if at least one access is a write.
- Enclave non-DRF regions do not access the same location as the matching xDRF region (at least one write).

4 Compile-Time Delineation of xDRF and xDRF-CI Regions

We implemented the automatic compile-time delineation of xDRF regions in LLVM [30] and integrated a state-of-the-art pointer analysis [31] to increase the accuracy. The algorithm of the xDRF and xDRF-CI analyses is illustrated below. Both analyses differ only in step 5.3 which deals with the conflict handling[1].

1) Identify synchronization points, i.e, nDRF regions (lock-unlock, atomics, join operations), and build the control-flow graph between them (Sync-CFG) (subsection 4.1).
2) Mark on the Sync-CFG the first reachable nDRF region (in depth-first-search order) in each thread function, as an entry nDRF region.
3) Identify nDRF regions that use the same synchronization variable (matching nDRF regions) (subsection 4.2).
4) Mark all join, barriers and signal-wait operations as non-enclave and the remaining nDRF regions as not-yet-processed (subsection 4.3).
5) Parse the Sync-CFG in a depth-first-search manner starting from each entry nDRF region. When unwinding, process each nDRF region as follows:
    - 5.1) Build the *preceding-xDRF-paths* and *following-xDRF-paths* for each nDRF region. Preceding- and following-xDRF-paths represent control-flow-paths that depart from the current nDRF region, extend across nDRF regions already marked as enclave and stop on the first encountered non-enclave or not-yet-processed nDRF region (subsection 4.4).
    - 5.2) Identify conflicts (subsection 4.5):
        - 5.2.1) Between the preceding-xDRF-paths and following-xDRF-paths of matching nDRF regions.
        - 5.2.2) Between the instructions within the matching nDRF regions and the preceding- and following-xDRF-paths.
        - 5.2.3) Between the instructions within any enclave nDRF crossed when building the xDRF paths of the matching nDRF regions (Step 5.2.1) and the preceding- and following-xDRF-paths of the current nDRF region.
    - 5.3a) (`xDRF`) If there are no conflicts, the nDRF region is enclave, otherwise non-enclave (subsubsection 4.5.1).
    - 5.3b) (`xDRF-CI`) If a conflict is detected (subsubsection 4.5.2):

1. Upon a conflict, the xDRF analysis marks the currently analyzed nDRF region as non-enclave. In contrast, the xDRF-CI analysis marks the currently analyzed nDRF region as enclave and isolates the conflicts.

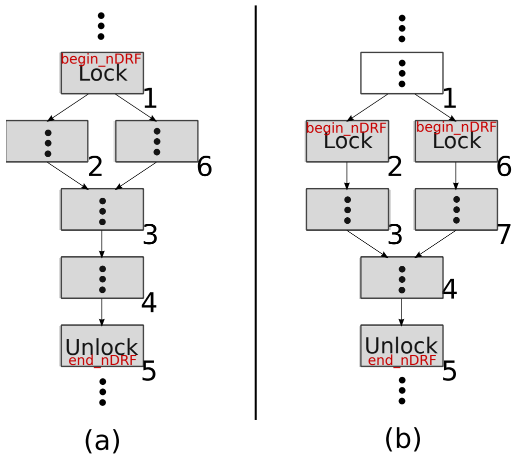


Fig. 4. Single-entry and multiple-entry nDRF regions: In (a) the depth-first-search reaches the lock operation in block 1 and starts a new nDRF region, collects blocks 2, 3 and 4, reaches block 5 and marks the end of the nDRF region and then backtracks and adds block 6. In (b) the control flow splits in block 1, before the start of the nDRF. Starting with the left branch, block 2 is recorded as the start of a new nDRF, the search adds blocks 3 and 4, then block 5 marking the end of the nDRF. When the search backtracks to block 1 and continues to block 6, block 6 will be recorded as the start of a second nDRF also including block 7. When discovering the visited block 4 of a *different* nDRF, the first and the second nDRFs are merged before backtracking.

5.3b.1) The currently analyzed nDRF region is marked as enclave.

5.3b.2) The conflicting memory accesses are each automatically guarded by one nDRF region.

5.3b.3) The newly inserted nDRF regions are marked as enclave. Their role is merely to isolate potentially shared data and to expose it to subsequent optimizations.

6) Parse the Sync-CFG in a depth-first-search manner starting from each entry nDRF region:

   6.1) If the current nDRF region is enclave, extend the current xDRF region, otherwise break the current xDRF region at this point and start a new one.

   6.2) If the current nDRF region is already enclave in another xDRF region, merge that xDRF region with the current xDRF region.

In what follows we detail each step of the xDRF and xDRF-CI delineation algorithm.

4.1 nDRF Region Delimitation

The analysis proceeds by *identifying synchronization points*, i.e. join operations, atomic instructions and regions of code guarded by acquire-release pairs[2]. We denote such regions non-DRF (in short, nDRF). To delineate nDRF regions, a depth-first-search is performed parsing the control-flow-graph (CFG), starting at the entry point of each thread (i.e. the function called by a newly spawned thread). The compiler delineates join operations, atomic instructions, or in the case of lock-unlock it marks the region in between as nDRF (CS in Figure 1). When a lock operation is encountered, it is recorded as the start of a new nDRF region. Instructions encountered on the depth-first-search path that follow the lock are added to the region up until the end of the nDRF region, i.e. the corresponding unlock operation, as shown in Figure 4. To handle nested or overlapping locks, a counter is held to make sure that all locks acquired within the nDRF region are released before the end. Lock-unlock pairs are matched by synchronization-variable in addition to the counter.

If an nDRF region has multiple starting points (Figure 4.b), i.e. a lock is acquired on two branches that later merge before the release of the lock, the depth-first-search algorithm will come across one lock operation before the other. An nDRF region will be created including the instructions between the encountered lock and unlock, however, instructions between the unvisited lock and the merge point of the two branches will not be detected at this time. Instead, when the depth-first-search algorithm naturally reaches the other lock operation, a new nDRF region will initially be recorded. When the algorithm eventually reaches the merge point of the branches, it detects that the next instruction has already been visited, like a regular depth-first-search algorithm would do, and also that it is part of a different nDRF region than the one being recorded. At this point the compiler can infer that the encountered nDRF region and the one being recorded is actually part of the same nDRF region, which causes the merging of the two records. Multi-exit regions are handled in the same way, merging regions that overlap: regions that depart from the same nDRF (single-entry) or regions that cross a block that has already been processed (multi-entry, multi-exit regions) are merged into one nDRF region.

In addition to the control-flow of each thread-function, we parse the call-graph and examine the callee functions to identify all synchronization points. The compiler keeps track whether the nDRF context extends inter-procedurally (the lock is acquired in the caller function and released in the callee). The analysis is full-path context-sensitive, with re-use of information from already analysed contexts. We handle recursions by collapsing the recursive call site to one point. We analyze whether a function is called from different contexts –(i) called from two nDRF regions using different locks or (ii) from one nDRF and one xDRF region, or if the function can only be called from the same context– (i) called from different nDRF regions but which synchronize on the same variable or (ii) it is only called from xDRF regions. We also handle functions called via indirection (function pointers), by conservatively analyzing all functions whose addresses are taken within the program. While parsing the control-flow and call-graphs, the analyses builds the Sync-CFG, a graph that records the control- and call-flow between all synchronization points (Step 1). For each thread function, starting from the entry block we analyze the control-flow path and the first encountered nDRF region is marked as an *entry-nDRF* (Step 2). Since Sync-CFG is a graph without a single root node, the entry nDRF regions will serve as starting points for subsequent analyzes on Sync-CFG.

4.2 Synchronization Variables of Matching nDRFs

To correlate nDRF regions that synchronize one with another, i.e. *matching nDRF regions*, we first identify the *synchronization variables* used by each nDRF region, namely expressions that can

2. In DRF applications, any synchronization point is either an atomic instruction or is guarded by an acquire-release pair. In the POSIX threads parallel programming paradigm, barriers, semaphores, signal-wait constructs, etc. are also implemented with or guarded by mutexes, i.e. lock-unlock operations (or atomics). Identifying synchronization points is the only part that is pthreads tailored, however the analysis can be easily extended to detect any other synchronization mechanisms.

be used for synchronization. To this end, all instructions of an nDRF region are analyzed and the synchronization instructions (pthread_mutex, pthread_condition, etc) are singled-out. The variables accessed by these instructions represent the synchronization values. For instance, in `call @pthread_mutex_lock(L)`, `L` is the synchronization value.

Starting from a synchronization value we build the set of variables this value aliases with and we denote this set a *synchronization variable*. Conservatively, synchronization values are in the same class if they MayAlias. Thus, synchronization variables are exhaustive, non-overlapping sets of synchronization values that (may) refer to the same shared variable. *Matching nDRF regions* are nDRF regions that share at least one synchronization variable (Step 3).

4.3 Pre-Analysis Marking of nDRFs

Before proceeding to analyze data sharing between threads in order to determine the nDRF regions' nature, we mark all join, barrier and signal-wait operations as non-enclave. The reasoning is that these synchronization points, by their semantics, impose the happens-before relation between threads due to data sharing.

In practice, in the xDRF delimitation, such operations would be marked as non-enclave, as shown by the manual markings. Based on this observation, pre-marking these nDRFs as non-enclave not only simplifies the analysis, but also alleviates the problem of identifying statically which threads synchronize through partial or indirect joins.

In xDRF-CI delineation, the most important aspect is the consequence on the analysis effectiveness: assuming that joins, barriers and signal-wait synchronization points do not represent xDRF boundaries, the entire application would become one large xDRF region. As the probability of aliasing increases over larger regions, the xDRF-CI would mark the majority of memory accesses as conflicting [3]. xDRF and xDRF-CI exploit the temporarily private nature of data within the xDRF regions.

The remaining nDRF regions are marked as not-yet-processed and their nature will be detected based on the data-sharing between threads, as described in what follows.

4.4 DRF and xDRF Paths

To determine the nature of each nDRF region, the compiler examines the *instructions on the control flow paths preceding the nDRF region* ($DRF1$ in Figure 1) and the *instructions following the nDRF region* ($DRF2$). The analysis builds the sets of instructions reachable *before* and *after* an nDRF region in two steps:

1) Collecting instructions on the DRF paths;
2) Collecting instructions on the xDRF paths;

Collecting instructions on the DRF-paths: We use the term *DRF-path* to denote a program path from one nDRF region to another, without passing through *any* nDRF region. The paths leading to a particular nDRF region are called the *preceding-DRF-paths* of the nDRF region, while the paths departing from a particular nDRF region are called the *following-DRF-paths* of that region. Figure 5(a), (b), and (c) shows examples of DRF paths for linear, divergent, and cyclic control-flow-graphs. The union of the preceding-DRF-paths builds the data-race-free region *before* the nDRF region of interest, while the union of the following-DRF-paths builds the data-race-free region *after* the nDRF region of interest. Note that the DRF-paths may have as limits different nDRF regions (Figure 5(b)) or that preceding- and following-DRF-paths may not be disjoint due to cycles in the control-flow-graph (Figure 5(c)).

Preceding- and following-DRF-paths are identified by parsing the CFG and the reverse-CFG, respectively, starting from the nDRF region of interest. The compiler collects instructions until an nDRF region is encountered, in which case the algorithm backtracks in search of not-yet-explored paths.

Collecting instructions on the xDRF-paths: Similarly, we use the term *xDRF-path* to denote a program path that starts from the current nDRF region, bypasses enclave nDRF regions (i.e. the xDRF path extends over enclave nDRF regions, but does not include the instructions from the critical section), until reaching a non-enclave or not-yet-explored nDRF region. Consequently, xDRF paths cannot bypass non-enclave nDRF regions. Akin to the notion of DRF paths, we use the terms *preceding-xDRF-paths* and *following-xDRF-paths* to refer to the union of xDRF paths leading to or starting from a given nDRF region. For example, in Figure 5(e), one preceding-xDRF-path contains blocks $\{1,2\}$, the other preceding-xDRF-path contains blocks $\{1,3\}$ and there is no other path leading to the current nDRF block. Thus, the union of the preceding-xDRF-paths contains the blocks $\{1,2,3\}$, denoted as xDRF-before in the figure. When building each xDRF-path, instructions on the xDRF paths are analyzed. Call instructions trigger the analyis of the callee functions, if their code is available. If library calls cannot be analyzed, the call instruction is conservatively marked as a non-enclave nDRF region. Library calls can also be white-listed (e.g. math operations, etc).

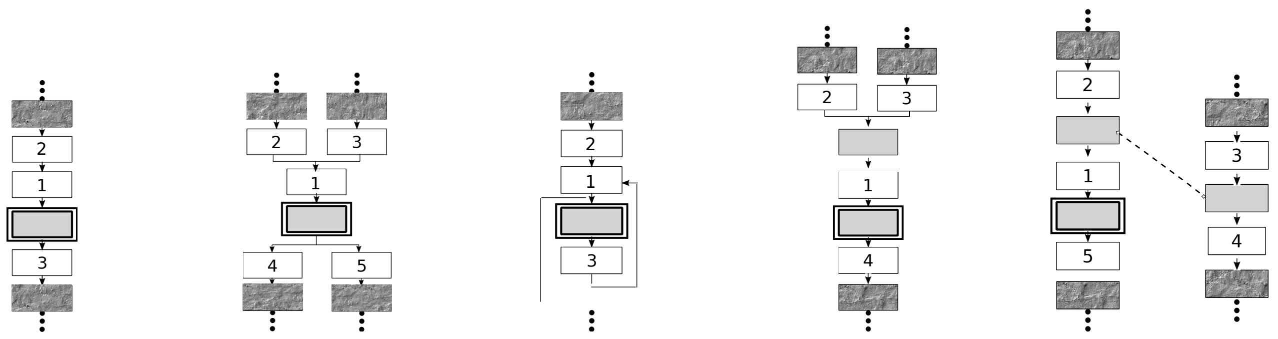
Function summaries are not preserved, instead functions are re-analyzed for each call. This ensures correct handling of functions called from different contexts: (i) called from two nDRF regions using different locks or (ii) from one nDRF and one xDRF region. One solution is to generate a new version per context (function cloning) or to use the most conservative of the classifications. To avoid code-size increase we took the latter approach and marked a synchronization-point as non-enclave if at least one context required it.

In what follows, we explain how the xDRF-paths are built. Before the nDRF regions are marked as enclave/non-enclave (i.e. not-yet-processed), the preceding- and following-xDRF-paths correspond the preceding- and following-**DRF**-paths of the current nDRF region, respectively. The approach is then to iteratively extend DRF-paths into xDRF-paths by confirming that the nDRF regions that synchronize the DRF paths can be enclave (subsection 4.5).

Figure 5(d) and (e) shows xDRF-paths that depart from the nDRF region of interest, *bypass nDRF regions already identified as enclave*, and continue the search on each path until a non-enclave nDRF region is encountered.

Furthermore, matching nDRF regions guide the analysis to other functions that synchronize on the same variable to model additional parts of the data-flow (Figure 2 (b-f)). Thus, the analysis "connects" the xDRF-paths of an enclave nDRF region to all xDRF-paths (both preceding- or following-xDRF paths) adjacent to a matching nDRF region. For instance, in Figure 5(e), the preceding-xDRF-path of the nDRF region of interest collects block 1, then crosses an enclave nDRF region and branches to the

3. As demonstrated in our previous work [12], [13], where we showed that techniques that classify data sharing over the entire execution (Operating-systems based classifications) classify almost all data as being shared.



(a) DRF-before={1,2} DRF-after={3}
(b) DRF-before={1,2,3} DRF-after={4,5}
(c) DRF-before={1,2,3} DRF-after={3,1}
(d) xDRF-before={1,2,3} xDRF-after={4}
(e) xDRF-before={1,2,3,4} xDRF-after={5}

Fig. 5. Examples of DRF and xDRF paths. The nDRF region of interest is shown as a double-bordered light-gray block. Dark-gray blocks represent nDRF regions already identified as non-enclave. Enclave nDRF regions are shown in light gray. White boxes represent basic blocks in a DRF region.

matching nDRF region (marked as connected by a dashed line). The search continues on the xDRF paths of each matching nDRF region until all paths have been explored. Searching on each xDRF path stops when a non-enclave block is reached. This algorithm adds blocks 2, 3 and 4 to the union of preceding-xDRF-paths.

4.5 Data Conflict Detection

Conflicts are detected from the perspective of each nDRF region, between three categories of accesses: (1) accesses on the preceding-xDRF-paths, (2) accesses on the following-xDRF-paths and (3) nDRF accesses (nDRF accesses refer to accesses from the current nDRF region and from its matching nDRF regions). A conflict occurs when accesses from different categories target the same memory location, at least one being a write. We denote such a conflict an *xDRF data conflict*. Note that conflicts do not occur between accesses of the same category. For instance, if accesses that belong to a preceding-xDRF-path incurred a conflict, this would be a regular data-race and the program would not be DRF. Regarding nDRF accesses, they are by definition synchronized and cannot lead to conflicts between them.

To determine if two accesses point to the same location, we complement traditional LLVM alias analysis [32] with a state-of-the-art pointer analysis [31] and report no conflict if at least one of the pointer analyses guarantees the accesses do not interfere.

Our analysis distinguishes between thread-local and variables that escape the thread scope and only checks for conflicts between variables visible to multiple threads (i.e. either global or escaped thread variables). This is implemented by tracing the def-use chain of the address in reverse order, searching for either a global variable, a function return, a function argument or a value that has been stored in or aliases non-local memory. We call such a value found on the def-use chain of the address a *shared value*. Given a pair of accesses, if at least one target address does not stem from a shared value, then there is no conflict between the two accesses. If each address stems from a shared value, but the shared values can be determined to be disjoint, then there is no conflict.

If the base addresses of two pointers stem from an escaped value (e.g. $global + off\_1$ and $global + off\_2$) we compare whether off_1 and off_2 can be equal, by tracking the dereferences (memory indirections) and offsets used in the pointer arithmetic. Furthermore, we discard aliases in which the offset is initialized with the thread ID. This simple extension can, in most cases, guarantee that accesses to different elements of data structures do not alias. Otherwise, a conflict is reported.

Matching nDRF regions can lead to transitive synchronization (recall Figure 2(f)). An xDRF path can "branch" to other paths on enclave nDRF regions in order to account that thread-ordering caused by synchronization is transitive, i.e. Th_0 and Th_1 may synchronize to establish a happens-before order and Th_1 and Th_2 synchronize as well, which implicitly synchronizes Th_0 and Th_2. Although the pairs of threads that synchronize explicitly might not share data (i.e. Th_0 and Th_1 or Th_1 and Th_2), the implicitly synchronized threads (Th_0 and Th_2) may share data outside critical sections as the DRF properties can be guaranteed by the happens-before order established by the synchronization points.

Conflicts between transitively synchronized threads are detected in multiple steps. In the example from Figure 2(f), in the first step, the compiler checks for conflicts between memory accesses on the preceding-xDRF-path of $signal\ A$ and on the following-xDRF-path of $wait\ A$ (and vice-versa) and no conflict is detected. In both xDRF and xDRF-CI delineations, this signal-wait pair will be now marked as "enclave" (see subsubsection 4.5.1 and subsubsection 4.5.2 below). In the second step, the compiler checks the synchronization point $signal\ B$. When collecting the memory accesses on the preceding-xDRF-path of $signal\ B$, the compiler encounters another synchronization point $wait\ A$ marked as "enclave". Therefore, it recursively collects all memory accesses on the xDRF-paths of $wait\ A$ and on the xDRF-paths of any matching nDRF region (i.e. the xDRF-paths of $signal\ A$), reaching the access $x = 1$ in Th_0, in conflict with the read of x ($print\ x$) in Th_2.

Cyclic xDRF paths (the preceding- and following-xDRF-paths overlap, as in Figure 5(c)) are handled as follows:

- Non-overlapping blocks of one xDRF-path are checked for conflicts against the other xDRF-path: block 2 from the preceding-xDRF-path and blocks 1 and 3 from the following-xDRF-path, in Figure 5(c);
- Blocks belonging to the loop are analyzed for loop carried dependences: blocks 1 and 3.

To expose region boundaries to the hardware, the compiler marks begin/end_xDRF, begin/end_nDRF regions through special instructions, akin [29,30]. Conflict handling represents the main difference between the xDRF and the xDRF-CI delineations, as shown below.

4.5.1 xDRF Region Annotation

In the xDRF delineation, in case a conflict is detected, the nDRF region separating the conflicting accesses (i.e. the currently analyzed nDRF region) is marked as non-enclave. Furthermore, if a conflict is detected between xDRF paths that cross nDRF regions already marked as enclave, not only the currently analyzed nDRF region is marked as non-enclave, but also the status of the nDRF region adjacent to the conflicting access is changed from enclave to non-enclave. This aspect is illustrated, in Figure 5(e), if a conflict is detected between block 5 and block 3, both the currently analyzed nDRF region and the one following block 3 are marked as non-enclave.

Similarly, in case of the transitive synchronization shown in Figure 2(f), once the conflict between Th_0 and Th_2 is detected (during the analysis of the signal-wait operations on B), the signal-wait operations on A are changed from enclave to non-enclave, and the signal-wait operations on B are marked as non-enclave. Thus, the signal-wait operations on A mark the boundaries of the xDRF regions in Th_0 and Th_1, while signal-wait on B mark the boundaries of the following xDRF regions in Th_1 and Th_2.

Once a synchronization was marked as an xDRF boundary (non-enclave), it cannot be promoted back to being "enclave".

4.5.2 xDRF-CI Conflict Isolation

To avoid fragmenting the xDRF regions due to false conflicts (i.e. memory accesses that may alias), we developed a technique which detects and isolates conflicts, without breaking the xDRF region (denoted as xDRF-CI). Once the conflicts are isolated, the synchronization points adjacent to the DRF regions that include the conflicting accesses are marked as enclave.

These nDRF regions are intended to isolate conflicts and include a single memory instruction. For full compatibility with the nDRF regions marked in our previous xDRF proposal, we decided to guard the conflicts with begin/end_nDRF instructions. Alternatively, a new class of memory operations forced to operate as nDRF instructions could be added, to reduce the number of instructions in applications with many isolated conflicts.

5 Formal Definitions and Proofs

5.1 Formal Definitions

We start by providing definitions for the notions that represent the building blocks of an xDRF region (data race, data-race-free accesses, data-race-free region) and finally we formally define the xDRF region and its properties.

Given the set of *conditions* on a pair of accesses:

$\textcircled{1}$ In a multi-threaded process, two accesses executed by different threads target the same memory location and at least one of the accesses is for writing;
$\textcircled{2}$ The accesses take place concurrently;
$\textcircled{3}$ At least one access is **not** a synchronization operation.

Definition 1: A *data race* occurs when all conditions hold: $\textcircled{1} \wedge \textcircled{2} \wedge \textcircled{3}$.

We denote two accesses a, b that incur a data race as $a \otimes b$.

Definition 2: Two accesses are called *data-race-free*, if at least one of the conditions do not hold: $\cancel{\textcircled{1}} \vee \cancel{\textcircled{2}} \vee \cancel{\textcircled{3}}$.

We denote two data-race-free accesses a, b as $a \bigcirc b$.

Corollary 1: In a multi-threaded process, if two data-race-free accesses can run concurrently, are not synchronization operations and at least one is a write operation, the accesses do not target the same memory location. Formally, $a \bigcirc b \wedge \textcircled{2} \wedge \textcircled{3} \Longrightarrow \cancel{\textcircled{1}}$.

Definition 3: A program $\mathcal{P}$ in which any pair of accesses is data-race-free is called data-race-free. Formally, $\mathcal{P}\ is\ DRF \Longleftrightarrow \forall a, b \in \mathcal{P}, a \bigcirc b$.

We denote that there is a path from an access a to an access b by $a \rightsquigarrow b$.

Definition 4: We denote a synchronization-race-free region (SFR) the set of instructions on the control-flow paths between two consecutive synchronizing operations, i.e., not including other synchronization operations on any path between them. Formally, $SFR\ region = \{instr | (instr \neq sync) \wedge (\forall a, b \in SFR, \nexists x, x = sync \wedge a \rightsquigarrow x \rightsquigarrow b)\}$, where $sync$ is a synchronization instruction.

We further divide SFR regions in two classes: (1) flexible, amenable to optimizations, such as regions between two unlock-lock operations and *excluding* the synchronization operations (i.e. outside critical sections), denoted as DRF regions; and (2) constrained, imposing restrictions, such as regions within two lock-unlock operations and *including* the synchronization operations (i.e. inside a critical section), denoted as $nDRF$. The step 1 in our algorithm identifies the $nDRF$ regions.

Definition 5: We define two nDRF regions $nDRF$ and $nDRF'$ as matching nDRF regions, and denote them by $nDRF \bowtie nDRF'$, if they synchronize on the same resource, i.e., use the same synchronization variable. Matching nDRF regions are identified in the step 3 of our algorithm.

We denote that the control flows from a region A to a region B as $A \rightarrowtail B$.

Definition 6: We define two DRF regions, DRF^A and DRF^B, as consecutive if for any path from DRF^A to DRF^B there is only one $nDRF$ region. Formally, $DRF^A \rightarrowtail nDRF \rightarrowtail DRF^B$.

Definition 7: We define as preceding-DRF-paths of an nDRF region $nDRF$ all accesses in a DRF region DRF such that $DRF \rightarrowtail nDRF$. Similarly, we define as following-DRF-paths of $nDRF$ all accesses in a DRF region DRF such that $nDRF \rightarrowtail DRF$.

Definition 8: An *xDRF path* is recursively defined as a single DRF path, or as the union of two $xDRF$ paths given the following conditions.

$$xDRF_{path} = \begin{cases} DRF_{path} \\ xDRF_{path} \bigcup xDRF_{path} \end{cases}$$

Let there be nDRF region $nDRF$, with its preceding-xDRF-paths DRF^A and its following-xDRF-paths DRF^B: $DRF^A \rightarrowtail nDRF \rightarrowtail DRF^B$.

And let there be any nDRF region $nDRF^{'}$ matching nDRF region $nDRF$, with its preceding-xDRF-paths $DRF^{A^{'}}$ and following-xDRF-paths $DRF^{B^{'}}$: $DRF^{A^{'}} \rightarrowtail nDRF^{'} \rightarrowtail DRF^{B^{'}}$, where $nDRF \bowtie nDRF^{'}$.

1) For any pair of accesses a, b , where a is on a preceding-xDRF-path (of either $nDRF$ or $nDRF^{'}$) and b is on a following-xDRF-path (of either nDRF region), the pair of accesses is data-race-free, $a \bigcirc b$ (step 5.2.1).
2) For any pair of accesses a, b , where a is on an xDRF-path ($\forall a \in \{xDRF^A,$ $xDRF^{A^{'}}, xDRF^B, xDRF^{B^{'}}\}$) and b is in an nDRF region ($\forall b \in \{nDRF, nDRF^{'}\}$), the pair of accesses is data-race-free, $a \bigcirc b$ (step 5.2.2).

The union of the preceding- ($xDRF^A$) and following-xDRF-paths ($xDRF^B$) of $nDRF$ build an xDRF-path ($xDRF^AB$) extending across $nDRF$. Similarly, preceding- and following-DRF-paths of $nDRF^{'}$ build an xDRF-path $xDRF^{A'B'}$ extending across $nDRF^{'}$.

Definition 9: The set of xDRF paths between consecutive non-enclave nDRF regions builds an xDRF region. Hence, an *xDRF region* is recursively defined as a single DRF region, or as the union of two $xDRF$ regions:

$$xDRF = \begin{cases} DRF \\ xDRF \bigcup xDRF \end{cases}$$

Definition 10: We define an nDRF region $nDRF$ as enclave in an xDRF region $xDRF^{AB}$, and denote as $nDRF \circledcirc xDRF^{AB}$, if $xDRF^A \rightarrowtail nDRF \rightarrowtail xDRF^B$ and $xDRF^A$ and $xDRF^B$ belong to $xDRF^{AB}$.

Definition 11: Two xDRF regions are said to be matching if they enclave matching nDRF regions. Formally, $xDRF \bowtie xDRF' \iff nDRF \circledcirc xDRF \wedge nDRF' \circledcirc xDRF' \wedge nDRF \bowtie nDRF'$.

The property of matching xDRF regions is transitive. If $xDRF$ matches $xDRF'$ and $xDRF'$ matches $xDRF''$ – possibly synchronizing on a different resource than the pair $(xDRF, xDRF')$– then no conflict can occur between the regions $xDRF$ and $xDRF''$ or between an xDRF region and the nDRF regions enclave in the "transitively matching" xDRF region (step 5.2.3). This property ensures that transitive synchronization and sharing of data between threads (see Figure 2(f)) is detected.

5.2 Proof of Correctness

5.2.1 xDRF analysis

The xDRF analysis (subsubsection 4.5.1) identifies non-enclave nDRF regions based on the observation that accesses in xDRF regions are data-race-free with accesses in other *concurrent* xDRF regions.

Corollary 2: In a multi-threaded DRF program, if two memory accesses target the same memory location, at least one is a write operation ($\textcircled{1}$) and they are not synchronization operations ($\textcircled{3}$), the accesses cannot run concurrently ($\cancel{\textcircled{2}}$). Formally, in a DRF application: $\textcircled{1} \wedge \textcircled{3} \implies \cancel{\textcircled{2}}$

Hence, Corollary 2 states that two accesses to the same memory location that are not synchronization must be separated by synchronization points that establish a *happens-before* order between the two accesses. On the premises that the input program is a data-race-free program, these synchronization points exist and represent boundaries of xDRF regions ($\textcircled{3}$). By detecting conflicts between matching xDRF regions, the xDRF analysis tests the nature of the synchronization point:

- If no conflict occurs, threads can access memory throughout the xDRF region in any relative order, without the need to communicate data (synchronize), except during the execution of enclave nDRF regions ($\cancel{\textcircled{1}} \wedge \textcircled{3} \implies \textcircled{2}$);
- If a conflict occurs, the synchronization is marked as non-enclave, thus, it is a boundary between the adjacent xDRF regions. Memory accesses performed in adjacent xDRF regions cannot be reordered across non-enclave nDRF regions, as they may access data shared between threads ($\textcircled{1} \wedge \textcircled{3} \implies \cancel{\textcircled{2}}$).

5.2.2 xDRF with Conflict Isolation

xDRF-CI regions preserve the DRF semantics across the entire region as defined by ***Definition 9***. By construction, xDRF-CI regions isolate conflicts by enclosing them in nDRF regions (step 5.3b.2). Since no conflicting accesses belong to the xDRF-CI, the nDRF regions separating the conflicts are marked as enclave (step 5.3b.1). Furthermore, following the same reasoning, the new nDRF regions are also enclave (step 5.3b.3).

As a consequence, nDRF regions that are marked as non-enclave in the xDRF delimitation (see Figure 3), are now marked as enclave in the xDRF-CI delimitation. While the xDRF delimitation prohibits instructions from adjacent xDRF regions to be reordered across these nDRF regions, the xDRF-CI delimitation refines this restriction and prohibits reordering *only for accesses that indeed should not cross the nDRF boundaries* (accesses to x in Figure 3). The conflicting accesses, enclosed in new nDRF regions, cannot migrate, since all nDRF regions preserve the original program order to ensure correctness. The remaining xDRF instructions (marked in green or blue blocks in Figure 3), can freely migrate across the boundaries of the nDRF regions, as long as dependences between instructions are respected.

To prove the correctness of the xDRF-CI delineation, consider two matching xDRF regions, $xDRF \bowtie xDRF'$ (according to ***Definition 11***) and two memory accesses $\forall a \in \{xDRF\},\ \ \forall b \in \{xDRF'\}$, at least one write, that are not synchronization operations $\textcircled{3}$. Since a and b belong to matching xDRF regions, they can execute concurrently $\textcircled{2}$. Assume that the xDRF-CI regions do not preserve the DRF semantics (*reductio ad absurdum*) and the two memory accesses conflict $a \otimes b$. By ***Definition 1***, $\textcircled{2} \wedge \textcircled{3} \implies \textcircled{1}$, which translates to accesses a and b target the same location and at least one is a write ($\textcircled{1}$). However, by construction of xDRF-CI, conflicting accesses are isolated as enclave nDRF regions (step 5.3b.3), which means that $a \notin \{xDRF\}$, and/or $b \notin \{xDRF'\}$, which contradicts our hypothesis. This proves the first property of xDRF regions, according to ***Definition 8*** and ***Definition 9***.

The same reasoning applies for any pair of accesses a, b, where a is on an xDRF-path ($\forall a \in \{xDRF\}$) and b is in an nDRF region ($\forall b \in \{nDRF\}$) enclave in the xDRF region, $nDRF \circledcirc xDRF$ (or in any matching $nDRF'$ region, $nDRF \bowtie nDRF'$). Assume again that the xDRF-CI regions do not preserve the DRF semantics (*reductio ad absurdum*) and the two memory accesses conflict $a \otimes b$. By the same reasoning as above, this implies that accesses a and b target the same location and at least one is a write ($\textcircled{1}$), which, by construction of xDRF-CI, would mark a as an nDRF region. This again contradicts our hypothesis that $a \in \{xDRF\}$. This proves the second property of xDRF regions, according to ***Definition 8*** and ***Definition 9***.

Thus, the xDRF-CI preserves the DRF semantics.

6 Evaluation

We divide the evaluation in two main parts. The first part focuses on the number of detected xDRF-CI regions, both from a static and from a dynamic perspective. The second part shows the impact of the xDRF-CI regions in SPEL++ [13], a cache coherence protocol optimized for DRF accesses.

Our evaluation is performed on applications from the Splash-3 [33] (a modernized, data-race-free version of the Splash-2 suite [34]) and Parsec-2.1 [35] benchmark suites. The applications are executed with the standard inputs for Splash-2/3 and simsmall inputs for Parsec-2.1. The run-time numbers are

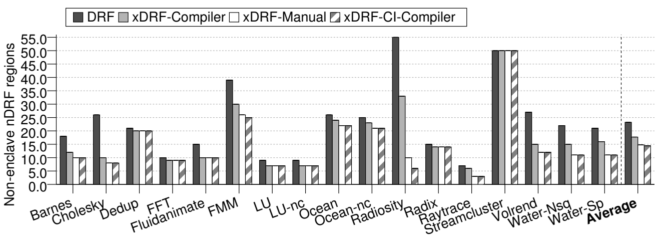


Fig. 6. Number of non-enclave nDRF regions in the code.

gathered after warming-up the caches in the parallel phase of the applications, as marked in the benchmark suites as regions of interest.

6.1 Large xDRF Regions with Conflict Isolation

We compare the proposed automatic compile-time delineation of large xDRF (labeled as *xDRF-CI-Compiler*) to our previous delineation of xDRF regions [14] (labeled as *xDRF-Compiler*[4]) and a manually delineation based on code inspection [36] (labeled as *xDRF-Manual*). Our baseline (labeled as *DRF*) is a DRF program where every nDRF region is considered as non-enclave, thus breaking the xDRF region. That is, the number of DRF regions in the baseline is equal to the number of xDRF regions.

We tested empirically that no data races occur in practice, by employing a consistency checker tool similar to Fast&Furious [37], extended to support xDRF regions. We run a hundred times each application for each delineation of xDRF regions and confirmed that no consistency violations appeared.

We perform first a static analysis for the entire application, and then we study the number of *executed* xDRF regions.

6.1.1 Static Large xDRF Regions

Figure 6 shows in the first bar (*DRF*) the number of nDRF regions in the code, all or them considered non-enclave. On average there are 23.2 nDRF regions per application. The automatic delineation of xDRF regions found that, on average, 5.5 of these regions can become safely enclave, thus merging the corresponding preceding and following xDRF regions into a single one.

There are still, however, 17.7 non-enclave regions, on average, in the *xDRF-Compiler* approach. Some of them (2.9 on average) are conservatively marked by the compiler as non-enclave, while the manual annotations performed with a careful code analysis showed that those regions can be safely enclave.

On average, the *xDRF-Compiler* approach misses optimization opportunities for more than 10% of the nDRF regions found in the code. It performs particularly well and matches the manual delineation on *Dedup*, *FFT*, *Fluidanimate*, *LU*, *LU-nc*, *Radix*, and *Streamcluster*. In contrast, other applications, such as *Barnes*, *Cholesky*, *FMM*, *Ocean*, *Ocean-nc*, *Radiosity*, *Raytrace*, *Volrend*, *Water-Nsq*, and *Water-Sp*, miss optimization opportunities due to the conservative approach of the compiler.

For instance, in *Barnes*, the compiler cannot identify that the conditional that guards a region of code ensures that only one thread can execute that region. An expert can reason about the semantics of the code in addition to detecting potential conflicts and manually mark the region as xDRF. *Radiosity*, *Raytrace*, and *FMM* operate on tasks that are obtained from a task-queue. Tasks are accessed via non-statically analyzable function pointers and the compiler cannot determine statically that each thread obtains a unique task. Assuming that each task can be executed by multiple threads, the compiler reports conflicts, whereas the expert can identify that the potentially conflicting accesses are actually performed by a single thread (i.e. each task is dequeued and executed by one thread only). Both *Water* benchmarks show the limits of the pointer analysis, as many of the may-alias conflicts reported by the compiler do not occur in practice. The conservatively non-enclave regions in *Cholesky* and *Raytrace* stem from a custom memory allocator which is called before the parallel region and within the region from the main thread. Similarly to *Barnes*, the compiler does not detect that only one thread actually can execute this code region and conservatively reports a potential conflict. *Cholesky* and *FMM* additionally report false-positives due to a recursive function forcing the same region to be checked for conflicts against itself (instructions may alias with themselves, even though each thread executes a different recursion levels of the function, similar to different iterations of a loop).

Thanks to conflict isolation, the *xDRF-CI-Compiler* can safely convert into enclave regions:

- The non-enclave regions marked **conservatively** by *xDRF-Compiler* (Incorrectly Non-Enclave nDRF regions), thus regaining missed optimization opportunities;
- The non-enclave nDRF regions marked **correctly** by *xDRF-Compiler*, but which can be handled as enclave if conflicting accesses are isolated (e.g. FMM, Radiosity).

Since we choose to apply conflict isolation only for nDRF regions which do not contain signal, broadcast or wait constructs, the last bar in Figure 6 shows the number of regions containing such constructs. On average, we reduce the number of non-enclave nDRF regions by 3.2 compared to our previous solution (*xDRF-Compiler*), with a peak of 27 less non-enclave nDRF regions in *Radiosity*.

The compilation time when automatically inserting xDRF and xDRF-CI annotations is shown in Table 1. The overhead of xDRF-CI over xDRF is negligible for all applications except FMM, Cholesky and Raytrace. Most applications have acceptable overhead when applying xDRF over the baseline. Cholesky and Radiosity, however, entail high overhead. Assuming that compilation-time becomes a concern, one can introduce a time-budget and stop the xDRF compile-time analysis once the given time-budget is exhausted. The experiments presented in the next section show that applications with a high-overhead compile-time analysis do not necessarily benefit from the xDRF analysis. Hence, the time-budget would be a viable solution, while still improving the performance of most applications.

6.1.2 Large xDRF Regions at Runtime

To understand the impact of xDRF regions, we conducted a study to count, at runtime, which regions are executed more frequently. Figure 7 complements Figure 6 by offering a runtime perspective. We plot the number of dynamic instances of non-enclave nDRF regions normalized to the number of executed nDRF regions. If the normalized bar equals 1 (i.e., all executed nDRF regions are non-enclave, which is equivalent to $\#xDRF = \#DRF$), this indicates that no DRF regions could be merged into the same xDRF region. In this figure lower is better, meaning that less but larger xDRF regions have been found.

4. We have fixed minor implementation issues in the previous version [14] which led to a few more non-enclave nDRF regions to be marked as enclave in *Dedup*, *Ocean*, *Ocean-nc*, *Radiosity*, and *Water-Sp* benchmarks.

TABLE 1
Compilation time in seconds.

Application	Base	xDRF	xDRF-CI
Barnes	1.928	8.383	8.352
Cholesky	10.228	2201.562	2046.017
Dedup	1.380	2.015	2.033
FFT	1.121	1.544	1.537
Fluidanimate	1.901	2.438	2.438
FMM	3.651	168.154	194.927
Lu	1.394	1.719	1.722
Lu-nc	1.663	2.081	2.081
Ocean	10.596	14.668	14.673
Ocean-nc	7.376	9.333	9.333
Radiosity	4.254	4636.695	4651.386
Radix	0.800	1.030	1.032
Raytrace	5.531	24.495	94.227
Volrend	3.514	29.561	24.409
Streamcluster	1.553	1.900	1.903
Water-Nsq	2.293	7.777	7.781
Water-Sp	2.549	5.697	5.882

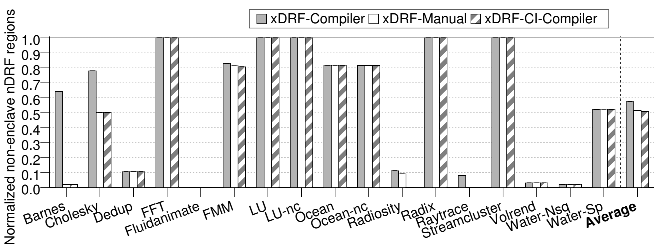


Fig. 7. Runtime number of non-enclave nDRF regions normalized to DRF.

The first bar in Figure 7 (*xDRF-Compiler*) shows the non-enclave nDRF instances of the automatic delineation. The second bar (*xDRF-Manual*) emphasizes the maximum potential without the conflict isolation technique. The third bar (*xDRF-CI-Compiler*) shows the results of using conflict isolation, which is the approach with less xDRF boundaries.

Some applications by their construction do not permit the merging of DRF regions into xDRF regions, since all or most of the nDRF regions are non-enclave. The reason is that they synchronize mainly based on barriers and signal-broadcast-wait constructs, which we treat as xDRF boundaries, even with the conflict isolation technique. Examples of these applications are *FFT*, *FMM*, *LU*, *LU-nc*, *Ocean*, and *Ocean-nc*, *Radix*, and *Streamcluster*, where most synchronization is based on barriers and signal-broadcast-wait constructs. Recall that dynamic instances are gathered on the regions of interest of the benchmarks, while static numbers are gathered over the entire application.

Applications with high potential (low number of static instances of non-enclave nDRF regions compared to the total number of nDRF regions in Figure 6 – e.g. *Cholesky*, *Radiosity* and *Volrend*), show, as expected, a low number of dynamic instances of non-enclave nDRF regions (Figure 7), i.e. large xDRF regions. This translates to a high potential when the xDRF analysis is employed for optimizations. Moreover, *Dedup*, *Fluidanimate* and *Raytrace*, contain a small number of static instances of enclave regions (Figure 6), but they are executed in a loop, yielding a large number of dynamic instances of enclave nDRF regions.

Barnes and *Cholesky* have some conservatively delineated non-enclave nDRF regions (xDRF-Compiler) with high impact at run time. Thanks to the conflict isolation technique these non-enclave regions are transformed into enclave, thus achieving the same number of xDRF regions as with the manual annotations.

Water-Nsq and *Water-Sp*, despite having also some conservative non-enclave regions in *xDRF-Compiler*, do not actually execute such nDRF regions in the region of interest of the application, hence there is no noticeable difference between the *xDRF-Compiler* and *xDRF-Compiler-CI* at run time.

6.2 Optimizing Coherence with large xDRF regions

Identifying xDRF regions offers great potential for optimizing cache coherence protocols. This section analyzes the impact of xDRF regions and large xDRF regions with conflict isolation in a state-of-the-art, dual-mode cache coherence protocol: SPEL++.

6.2.1 SPEL++: A Dual-Mode Cache Coherence Protocol

SPEL++ [13] deactivates coherence for memory accesses performed within xDRF regions and maintains traditional directory coherence for accesses within nDRF regions. Data accessed during xDRF regions are made visible (coherent with other threads) in the boundaries of xDRF regions (i.e. non-enclave nDRF regions), by flushing blocks cached privately. While nDRF memory references are resolved as in a standard directory protocol, accesses within xDRF regions perform in the following way:

- *Read misses*: Read misses obtain the data as in a directory protocol. The data block is stored in the cache in "private" mode without being tracked by the directory (the copy is invisible to the coherence protocol), making a more efficient use of its storage.
- *Write misses*: Store operations do not cause write misses nor invalidation messages, since they do not require read or write permission. Every store allocates space in cache and writes the new value. The block is marked as "private" and "dirty" bits are set to track every written byte.
- *Cache evictions and flushing*: A cache eviction of a "private" block has the same effect as a flush, employed to enforce coherence in the xDRF region boundaries. Clean blocks can be silently evicted. Dirty blocks require a write-back of the modified bytes. In case there are coherent copies of a block cached by remote cores (by an nDRF access), they should be first invalidated, and then updated with the data being written back.

6.2.2 Simulation Methodology

We employ the GEMS simulator [38], a detailed simulator for multiprocessor systems. GEMS is fed with information gathered by a Pin tool [39], which offers detailed information about the execution of the applications, such as executed instructions, memory references, and synchronization primitives. Synchronization primitives are captured by Pin and do not provide to the simulator the instructions executed, but the functional behavior of the primitive. This functional behavior is properly modeled in the internals of the simulator by generating the necessary instructions dynamically. The interconnection network has been modeled with GARNET [40], included in the GEMS toolset. Reported energy consumption has been obtained with McPAT [41], assuming a 32nm process technology.

We compare the different xDRF alternatives analyzed in the previous section on top of the SPEL++ cache coherence protocol and normalize them to a traditional directory protocol with MESI states (*Directory*). These cache coherence protocols have been

TABLE 2
System parameters.

Parameter	Value
Cache hierarchy	Non-inclusive
Block / Page size	64 bytes / 4 KB
Split instr & data L1 caches	32 KB, 8-way (128 sets)
L1 cache hit time	1 (tag) and 2 (tag+data) cycles
Shared unified L2 cache	512 KB / tile, 16-way (512 sets)
L2 cache hit time	6 (tag) and 12 (tag+data) cycles
Directory cache	64 sets, 8 ways (×1 L1)
Directory cache hit time	2 cycle
Memory access time	160 cycles
Topology	Mesh 2D
Flit size, link time	16 bytes, 1 cycle

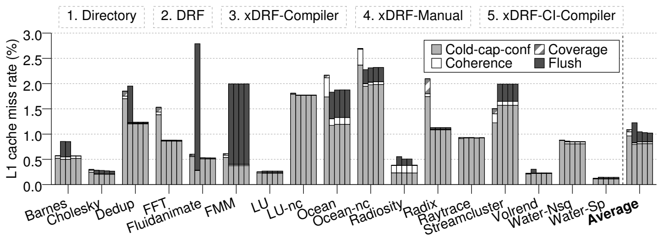


Fig. 8. L1 cache miss rate (%) classified by the cause of the miss.

modeled in detail using the SLICC domain specific language provided by GEMS. Since SPEL++ targets large-scale systems where the benefits of deactivating the coherence protocol are greater, we model a system with 64 in-order cores, private L1 caches, and shared L2 cache. Table 2 shows the remaining parameters of the simulated system.

6.2.3 Performance Results

Our goal is to build large xDRF regions, since larger xDRF regions imply a lower number of region boundaries, and consequently less flush operations required to keep coherence of DRF accesses. Since a flush operation invalidates and/or downgrades the private content of the cache, reducing the number of flush operations results in less invalidations and write-backs, and thus less cache misses and coherence traffic. We first focus on the impact of cache misses and then on the impact on network traffic. Finally, we show their implications on execution time and energy consumption.

Figure 8 shows the L1 cache miss rate (in percentage) for the five configurations evaluated. The miss rate has been split in several categories, indicating the cause that generated the miss. The first bar, which illustrates the miss rate in a directory protocol, is split in the *5C* classification of misses [42]: (i) cold or compulsory, capacity, and conflict misses (*Cold-cap-conf*, or *3C*); (ii) coherence misses (*Coherence*), as a consequence of invalidations and downgrades generated by remote writes and reads, respectively; and (iii) misses that stem from invalidations generated by directory evictions (*Coverage*). The following bars show SPEL++ (with different delineations of xDRF regions) which adds an extra category of misses due to a partial cache self-invalidation caused by flush operations (*flush*).

From Figure 8 we can observe that SPEL++ with a simple DRF delineation causes an increase of misses compared to directory, on average, and in particular in *Barnes*, *Dedup*, *Fluidanimate*, *FMM*, *Radiosity*, *Volrend* and *Streamcluster*. The automatic xDRF delineation can avoid most of these misses, reducing the miss rate compared to directory due to the removal of most coherence

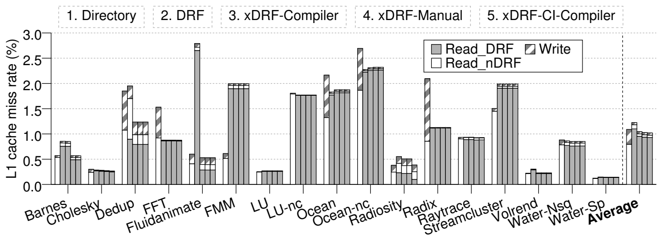


Fig. 9. L1 cache miss rate (%) classified by the type of the miss.

and coverage misses. However, on average, in xDRF, 19.1% of the cache misses are due to flush operations, while the manual delineation reduces these misses to 17.6%. The additional misses (compared to Manual) are due to extra flush operations that appear because of the conservative automatic analysis. In particular, *Barnes* is an example where the automatic delineation is far from the manual one. When isolating the conflicts within enclave nDRF regions, the number of flush operations is reduced and only 17.0% of the misses are due to crossing of xDRF boundaries . Both *Barnes* and *Radiosity*, are clear examples where significant improvements are obtained with respect to manual, thanks to the isolation of conflicts.

Figure 9 shows the same miss rate as in Figure 8, but now showing how each miss is resolved. Writes in DRF regions are always considered as hits, and this is the reason they do not appear in the figure. Read misses in directory are always treated as nDRF, while SPEL++ saves coherence traffic by executing misses within DRF regions in DRF-mode. If conflict isolation (*CI*) would isolate also across barriers and signal/wait, the number of read_nDRF would increase in SPEL++, going back to directory numbers. There are however two applications, namely *FMM* and *Streamcluster*, where the miss rate increases when applying the CI technique with SPEL++, compared to directory. In such cases it would be interesting to analyze more aggressive conflict resolution policies, i.e., across signal/wait or barriers. The downside of such policy is a potential high number of nDRF misses. This trade-off analysis is left as future work.

As mentioned, larger xDRF regions help to reduce coherence traffic. Figure 10 shows the network traffic generated by the applications, normalized to *Directory*. SPEL++ for DRF regions is able to reduce the network traffic in most cases, except in *Barnes*, *Fluidanimate*, *FMM*, and *Radiosity*. These are synchronization-intensive applications, and therefore contain a large number of small DRF regions. This leads to many cache flushes which cancels the benefits of SPEL++ for DRF codes. On the other hand, the automatic identification of xDRF regions reduces noticeably the traffic in *Fluidanimate*, *Dedup*, *Radiosity*, and *Volrend*, as a consequence of merging DRF regions into larger xDRF regions (see Figure 7). On average, the network traffic is reduced by 20.7% with *xDRF-Compiler*. Furthermore, when applying the conflict isolation technique, the traffic can be reduced by 23.1%, mainly due to the reductions in *Barnes* and *Radiosity*.

Figure 11 shows the execution time normalized to a directory protocol. Again, we observe that, on average, SPEL++ with mere DRF delineation is on-par or slightly out-performed by the baseline. When applying automatic xDRF delineation, execution time is reduced by 10.0% (compared to DRF), leading to 6.8% improvements with respect to *Directory* and almost on-par with the ideal, manual delineation (8.1%). The only exception is *Barnes*,

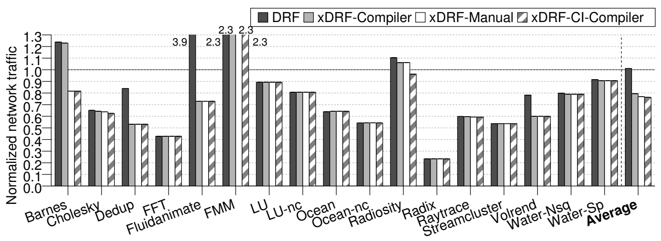


Fig. 10. Network traffic normalized to *Directory*.

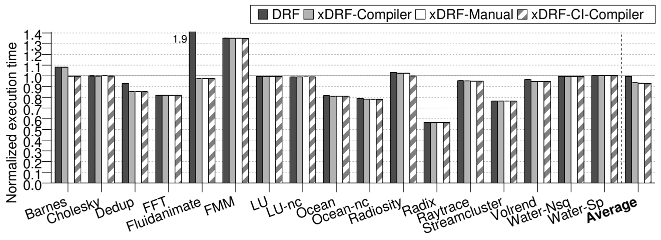


Fig. 11. Execution time normalized to *Directory*.

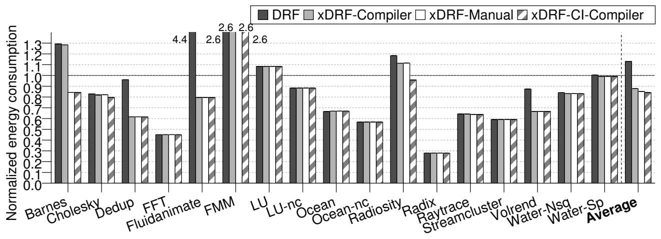


Fig. 12. Energy consumption normalized to *Directory*.

where the automatic delineation does not reach the performance of the manual one. Thanks to the conflict resolution technique, xDRF-CI can achieve numbers on par with the manual delineation (*Barnes*) and even outperform it (*Radiosity*).

Finally, Figure 12 plots the energy expenditure normalized to *Directory*. Clear improvements are observed when using the xDRF delimitation compared to both *DRF* and *Directory*. On average, the automatic compile-time delineation with conflict resolution is able to save 15.9% of the energy consumed by a directory protocol.

7 Conclusions

We describe an automated compile-time classification of "unmanaged" parallel programs which delineates DRF regions and identifies extended data-race-free regions (xDRF) and large extended data-race-free regions with conflict isolation (xDRF-CI). xDRF and xDRF-CI are regions of code that bypass and extend across synchronization points (acquire-release pairs), loop backedges, function calls, etc, and guarantee data-race-freedom semantics, similar to one large synchronization-free region. The conflict isolation technique allows to extend even further the xDRF regions alleviating the drawbacks that xDRF introduces for some applications and matching the performance or even outperforming the manually annotated version.

Acknowledgments

This work was funded by the Swedish Research Council under the VR Starting Grant 2016-05086. The work was also supported by the Spanish MINECO, as well as European Commission FEDER funds, under grant TIN2015-66972-C5-3-R and the Fundación Séneca under the project "Jóvenes Líderes en Investigación" 18956/JLI/13.

References

[1] S. V. Adve and H.-J. Boehm, "Memory models: A case for rethinking parallel languages and hardware," *Communications of the ACM*, vol. 53, no. 8, pp. 90–101, Aug. 2010.

[2] ISO, *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization, 2011.

[3] ——, *ISO/IEC 14882:2015 Information technology — Programming languages — C++*. International Organization for Standardization, 2015.

[4] T. J. Ashby, P. Díaz, and M. Cintra, "Software-based cache coherence with hardware-assisted selective self-invalidations using bloom filters," *IEEE Transactions on Computers (TC)*, vol. 60, no. 4, pp. 472–483, Apr. 2011.

[5] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, "DeNovo: Rethinking the memory hierarchy for disciplined parallelism," in *20th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2011, pp. 155–166.

[6] H. Sung, R. Komuravelli, and S. V. Adve, "DeNovoND: Efficient hardware support for disciplined non-determinism," in *18th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2013, pp. 13–26.

[7] H. Sung and S. V. Adve, "DeNovoSync: Efficient support for arbitrary synchronization without writer-initiated invalidations," in *15th Int'l Conf. on Architectural Support for Programming Language and Operating Systems (ASPLOS)*, Mar. 2015, pp. 545–559.

[8] A. Ros and S. Kaxiras, "Complexity-effective multicore coherence," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 241–252.

[9] S. Kaxiras and A. Ros, "A new perspective for efficient virtual-cache coherence," in *40th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2013, pp. 535–547.

[10] M. Elver and V. Nagarajan, "RC3: Consistency directed cache coherence for x86-64 with RC extensions," in *24th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2015, pp. 292–304.

[11] S. V. Adve and M. D. Hill, "Weak ordering – a new definition," in *17th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1990, pp. 2–14.

[12] A. Ros and A. Jimborean, "A dual-consistency cache coherence protocol," in *29th Int'l Parallel and Distributed Processing Symp. (IPDPS)*, May 2015, pp. 1119–1128.

[13] ——, "A hybrid static-dynamic classification for dual-consistency cache coherence," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 27, no. 11, pp. 3101–3115, Nov. 2016.

[14] A. Jimborean, J. Waern, P. Ekemark, S. Kaxiras, and A. Ros, "Automatic detection of extended data-race-free regions," in *15th IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Feb. 2017, pp. 14–26.

[15] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti, "A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code," in *38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, Jan. 2011, pp. 623–636.

[16] L. Effinger-Dean, H.-J. Boehm, D. Chakrabarti, and P. Joisha, "Extended sequential reasoning for data-race-free programs," in *2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC)*, Jun. 2011, pp. 22–29.

[17] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, "Ifrit: Interference-free regions for dynamic data-race detection," in *2012 ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2012, pp. 467–484.

[18] A. Singh, S. Narayanasamy, D. Marino, T. Millstein, and M. Musuvathi, "End-to-end sequential consistency," in *39th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2012, pp. 524–535.

[19] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-assisted data distribution for chip multiprocessors," in *19th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2010, pp. 501–512.

[20] Y. Li, R. G. Melhem, and A. K. Jones, "Practically private: Enabling high performance cmps through compiler-assisted data classification," in *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2012, pp. 231–240.

[21] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," in *22th ACM Symp. on Operating Systems Principles (SOSP)*, Oct. 2003, pp. 237–252.

[22] M. Abadi, C. Flanagan, and S. N. Freund, "Types for safe locking: Static race detection for java," *ACM Transactions on Programming Languages and Systems (TPLS)*, vol. 28, no. 2, pp. 207–255, Mar. 2006.

[23] M. Naik, A. Aiken, and J. Whaley, "Effective static race detection for java," in *2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2006, pp. 308–319.

[24] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," in *2014 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Mar. 2014, pp. 337–348.

[25] C. Flanagan and S. N. Freund, "Fasttrack: Efficient and precise dynamic race detection," in *2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2009, pp. 121–133.

[26] B. Lucia, L. Ceze, K. Strauss, S. Qadeer, and H.-J. Boehm, "Conflict exceptions: Simplifying concurrent language semantics with precise hardware exceptions for data-races," in *37th Int'l Symp. on Computer Architecture (ISCA)*, Jun. 2010, pp. 210–221.

[27] X. Xie and J. Xue, "Acculock: Accurate and efficient detection of data races," in *9th IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Apr. 2011, pp. 201–212.

[28] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia, "Valor: Efficient, software-only region conflict exceptions," in *15th ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, Oct. 2015, pp. 241–259.

[29] B. Hackett and A. Aiken, "How is aliasing used in systems software?" in *Proceedings of the Symposium on the Foundations of Software Engineering*, 2006, pp. 69–80.

[30] C. Lattner and V. S. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *2nd IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Mar. 2004, pp. 75–88.

[31] Y. Sui, P. Di, and J. Xue, "Sparse flow-sensitive pointer analysis for multithreaded programs," in *14th IEEE / ACM Int'l Symp. on Code Generation and Optimization (CGO)*, Mar. 2016, pp. 160–170.

[32] "LLVM Alias Analysis," website, Mar. [Online]. Available: http://llvm.org/docs/AliasAnalysis.html

[33] C. Sakalis, C. Leonardsson, S. Kaxiras, and A. Ros, "Splash-3: A properly synchronized benchmark suite for contemporary research," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2016, pp. 101–111.

[34] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *22nd Int'l Symp. on Computer Architecture (ISCA)*, Jun. 1995, pp. 24–36.

[35] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *17th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Oct. 2008, pp. 72–81.

[36] A. Ros, C. Leonardsson, C. Sakalis, and S. Kaxiras, "Poster: Efficient self-invalidation/self-downgrade for critical sections with relaxed semantics," in *25th Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2016, pp. 433–434.

[37] A. Ros and S. Kaxiras, "Fast&furious: A tool for detecting covert racing," in *6th Workshop on Parallel Programming and Run-Time Management Techniques for Many-core Architectures (PARMA) and 4th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (DITAM)*, Jan. 2015, pp. 1–6.

[38] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *Computer Architecture News*, vol. 33, no. 4, pp. 92–99, Sep. 2005.

[39] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *2005 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, Jun. 2005, pp. 190–200.

[40] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in *IEEE Int'l Symp. on Performance Analysis of Systems and Software (ISPASS)*, Apr. 2009, pp. 33–42.

[41] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *42nd IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, Dec. 2009, pp. 469–480.

[42] A. Ros, B. Cuesta, R. Fernández-Pascual, M. E. Gómez, M. E. Acacio, A. Robles, J. M. García, and J. Duato, "EMC^2: Extending magny-cours coherence for large-scale servers," in *17th Int'l Conf. on High Performance Computing (HiPC)*, Dec. 2010, pp. 1–10.

Alexandra Jimborean is Assistant Professor at Uppsala University since 2015. She obtained her PhD from the University of Strasbourg, France in 2012, was awarded the Anita Borg Memorial Scholarship offered by Google in recognition of excellent research, along with other 25 distinctions, awards and grants. Her research focuses on compile-time and run-time code analysis and optimization for performance and energy efficiency and on software-hardware co-designs.

Per Ekemark is an undergraduate student of Computer Science at Uppsala University since 2011. His bachelor's thesis explored compile-time performance and energy efficiency optimizations and was featured in an award-winning international publication. His research interest are compile-time optimizations and concurrent and parallel programming.

Jonatan Waern is a software developer at Intel Windriver working on internal technologies. Previously he studied computer science at Uppsala University, with a specialization in compilers and parallel programming.

Stefanos Kaxiras is a full professor at Uppsala University, Sweden. He holds a PhD degree in Computer Science from the University of Wisconsin. Previously he held positions at Bell Labs (Lucent) and the University of Patras, Greece. His research interests are in the areas of memory systems, and multiprocessor/multicore systems, with a focus on power efficiency. He is a Distinguished ACM Scientist and IEEE member.

Alberto Ros is Associate Professor at the University of Murcia, Spain. He received the PhD degree in computer science from the same university, in 2009, after being granted with a fellowship from the Spanish government. He hold postdoctoral positions at the Universitat Politècnica de València and at Uppsala University. His research interests include cache coherence protocols, memory hierarchy designs, and memory consistency for multicore architectures.