Automatic Detection of Extended Data-Race-Free Regions

Alexandra Jimborean Jonatan Waern Per Ekemark Stefanos Kaxiras Alberto Ros*

Uppsala Universitet, Sweden *Universidad de Murcia, Spain

alexandra.jimborean@it.uu.se, stefanos.kaxiras@it.uu.se, aros@ditec.um.es

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 regions (xDRF)*, namely regions of code which 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. Our compiler techniques precisely analyze the threads' memory accessing behavior and data sharing in shared-memory, general-purpose parallel applications and can therefore infer the limits of xDRF code regions.

We evaluate the potential of our technique by employing the xDRF region classification in a state-of-the-art, dual-mode cache coherence protocol. Larger xDRF regions reduce the coherence bookkeeping and enable optimizations for performance (6.8%) and energy efficiency (11.7%) compared to a standard directory-based coherence protocol.

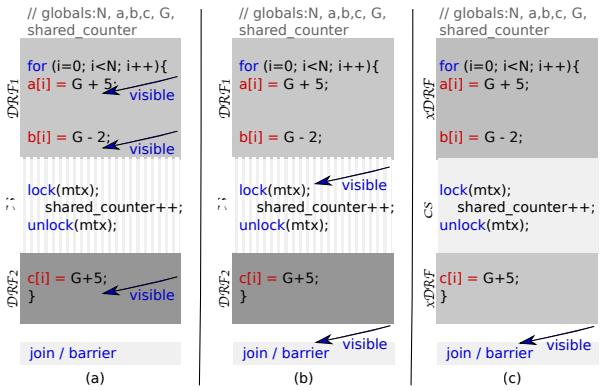


Figure 1. (a) One loop iteration contains two data-race-free regions (DRF1 and DRF2), interleaved with a critical section (CS). A standard coherence protocol makes the $store$ operations visible immediately after they have executed, thus performing $3\times$N actions. (b) Coherence protocols designed for DRF applications delay the action of making write operations visible until the first encountered synchronization point, hence $N+1$ actions. (c) The xDRF regions consists of both DRF1 and DRF2 regions (bypassing CS). An optimized 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 1.

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 adopt 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 [3, 16, 17]. 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 [6, 9, 12, 19, 31, 37, 38]. 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. Similarly, C/C++ compilers and alike typically optimize synchronization-free regions as if the code was sequential, without the need for speculation or costly inter-thread analysis.

In this paper we denote synchronization-free regions that are *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 [4] across the entire region [29, 30]. For example, in Figure 1.c, the

978-1-5090-4931-8/17 © 2017 IEEE

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*). Assuming that threads execute different iterations of the loop, the memory locations modified during the xDRF region are distinct —$a[i] \neq b[i] \neq c[i]$, $a[i] \neq a[j]$ if $i \neq j$ etc— and do not alias with read-only global variables G, N. Furthermore, $shared\_counter$, read and modified in the critical section, points to a different location than the other global variables.

In short, xDRF regions enable optimizations across synchronization 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-based 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. Previous work emphasized the benefits of xDRF regions, showing significant performance and energy improvements for automatically parallelized and OpenMP applications [29, 30]. 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.

This work aims to increase the applicability of the xDRF delineation to a considerably larger class of applications. We propose compiler techniques to statically identify xDRF regions in "unmanaged" shared-memory parallel applications that follow the fork-join with synchronization model, such as pthreads-based parallel applications. 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) [29, 30], we target applications where the programmer, not the compiler, has control over the way parallelism is expressed.

Departing from DRF applications, we propose a compile-time technique for identifying xDRF regions. To verify the xDRF properties, the compiler analyzes aliasing not only between memory accesses initiated from data-race-free regions —$a[i]$, $b[i]$, $c[i]$, G, N— but must also compare whether these accesses alias the memory locations accessed from critical sections ($shared\_counter$). Next, the inter-thread, inter-procedural analysis crosses threads accessing the **same synchronization variables** and verifies that the memory accesses **before** the critical section of one thread do not conflict with the accesses **after** the matching critical section of another thread.

This work makes the following contributions:

1. We automatically identify xDRF regions in general-purpose parallel applications that follow the fork-join with synchronization model (POSIX-threads). xDRF is an inter-thread and inter-procedural analysis, which: (i) Extends across synchronization points and even *across multiple acquire-release pairs*. (ii) Is full-path context-sensitive, where contexts are defined with respect to synchronization regions: we distinguish between *matching synchronization points* (threads using the same synchronization variable), *non-matching synchronization points* (threads using different synchronization variables) and xDRF regions (outside critical sections) and reason about their semantics for delimiting xDRF regions. (iii) Extends the synchronization-free guarantees to a larger scope.
2. We formalize the definition of xDRF regions and the correctness of our static analysis.

We evaluate the potential of the xDRF classification with a state-of-the-art, dual-mode cache coherence protocol [30] 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 (6.8%) and energy efficiency (11.7%) compared to a standard directory-based protocol.

2. What Are xDRF 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, Th_1 prints its value, hence the barrier represents a limit between two different xDRF regions. The first xDRF region contains the initialization $x = 1$ and the second xDRF region prints x. 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 *en-*

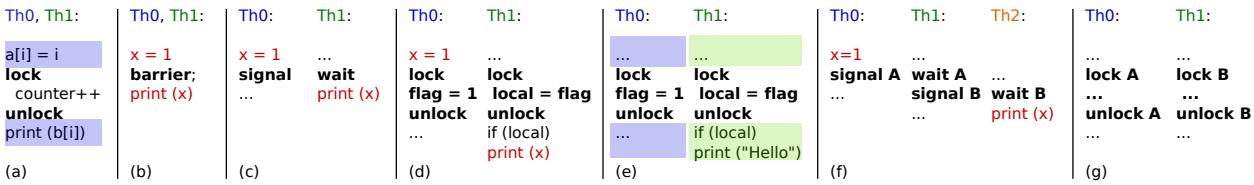


Figure 2. In (a) and (b) multiple threads execute the same code and cross a synchronization point, while (c) - (f) show examples when the threads execute different code regions that synchronize on the same resource. In (g) threads do not lock the same resource, hence the regions are not synchronized and do not contribute to the xDRF delimitation of the other thread. Conflicts are marked in red 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).)

clave (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. 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) are not synchronized 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 within the critical section the instruction $if(local)print(x)$ performed by T_1. 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.

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. And, by extension, xDRF regions corresponding to different threads are called *matching xDRF regions*, if they enclave matching nDRF regions. 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).

3. Compile-Time Delineation of xDRF Regions

We implemented the automatic compile-time delineation of xDRF regions in LLVM [20] and integrated a state-of-the-art pointer analysis [36] to increase the accuracy. The algorithm of the xDRF analysis consists of the following steps:

1. Identify synchronization points – nDRF regions – (lock-unlock, atomics, join operations) and build the control-flow graph between them (Sync-CFG) (subsection 3.1).
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 3.2).
4. Mark all join, barriers and signal-wait operations as non-enclave and the remaining nDRF regions as not-yet-processed (subsection 3.3).

5. Parse the Sync-CFG in a depth-first-search manner starting from each entry nDRF region. When unwinding, determine for each nDRF region if it is enclave, 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 3.4).
   5.2. Identify conflicts (subsection 3.5):
      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.
   5.3. If there are no conflicts, the nDRF region is enclave, otherwise non-enclave (subsection 3.5).
6. The nature of each nDRF region (enclave or non-enclave) automatically determines the xDRF boundaries. Enclave regions permit the xDRF region to extend across them, while non-enclave regions represent boundaries between adjacent xDRF regions. The starting xDRF region for each thread is the trivial DRF region leading up to the first nDRF region. Next, 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 delineation algorithm.

3.1 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[1]. 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 3. 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.

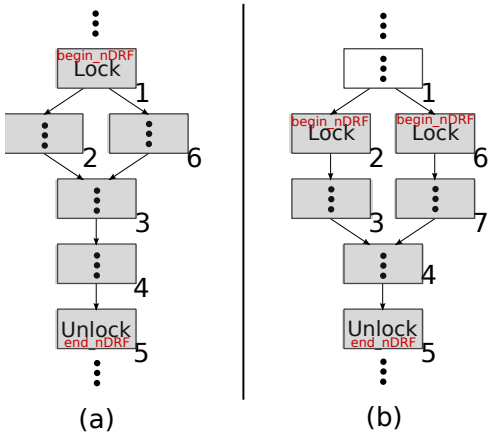


Figure 3. Single-entry and multiple-entry nDRF regions: In (a) the depth-first-search reaches the lock operation in block 1 and starts a new nDRF region, collects blocks 2, 3 and 4, reaches block 5 and marks the end of the nDRF region and then backtracks and adds block 6. In (b) the control flow splits 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 ads 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.

If an nDRF region has multiple starting points (Figure 3.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

[1] 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.

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.

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.

3.2 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 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 single-out. The variables accessed by these instructions represent the synchronization values. For instance, in `call @pthread_mutex_lock(lock)`, `lock` 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. Therefore, synchronization variables are exhaustive, non-overlapping sets of synchronization values that (may) refer to the same global variable. *Matching nDRF regions* are nDRF regions that share at least one synchronization variable (Step 3).

3.3 Pre-Analysis Marking of nDRFs

Before proceeding to the analysis of data sharing between threads in order to mark nDRF regions as enclave or non-enclave, 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. Since in practice such operations indeed almost always are marked as non-enclave, we simplify the analysis using this conservative pre-analysis step. Furthermore, by marking these nDRFs as non-enclave, we alleviate the problem of identifying statically which threads synchronize and handling of partial and indirect joins.

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.

3.4 DRF and xDRF Paths

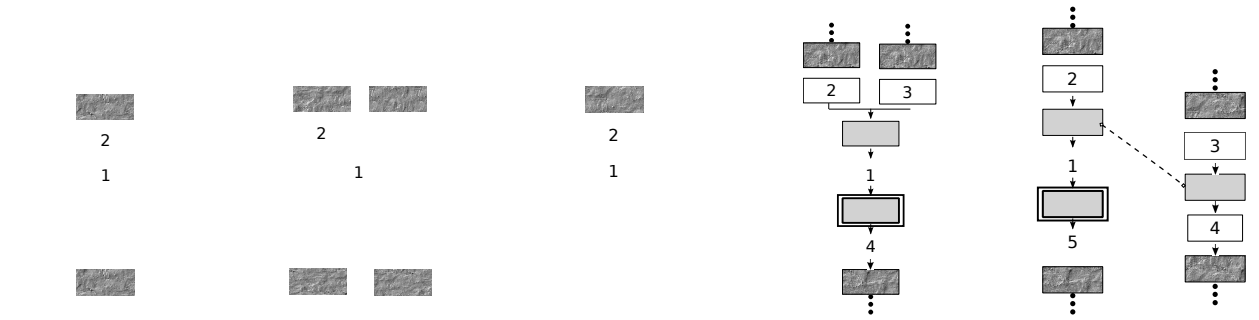
To determine the nature of each nDRF region, the compiler examines the *instructions on the control flow paths preceding the nDRF region* ($DRF1$ in Figure 1) and the *instructions following the nDRF region* ($DRF2$). The xDRF analysis builds the sets of instructions reachable *before* and *after* an 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 4(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 4(b)) or that preceding- and following-DRF-paths may not be disjoint due to cycles in the control-flow-graph (Figure 4(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 4(e),



(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}

Figure 4. 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.

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 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 3.5).

Figure 4(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 4(e), the preceding-xDRF-path of the nDRF region of interest collects block 1, then crosses an enclave nDRF region and branches to the 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.

3.5 xDRF 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 [1] with a state-of-the-art pointer analysis [36] 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 function and only checks for conflicts between variables visible to all threads (i.e. either global or escaped thread-local 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 de-references (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 and the nDRF region separating them (currently analyzed nDRF region) is marked as non-enclave. When 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. For example, in Figure 4(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.

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. This signal-wait pair is now marked as "enclave". 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 $x = 1$. The conflict between Th_0 and Th_2 is therefore detected and the signal-wait operations on A is changed from enclave to non-enclave, marking the boundary of the xDRF regions (in Th_0 and Th_1), while signal-wait on B marks the boundary 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".

Cyclic xDRF paths (the preceding- and following-xDRF-paths overlap, as in Figure 4(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 4(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].

4. Formal Definitions and Proofs

4.1 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:

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;
② The accesses take place concurrently;
③ At least one access is **not** a synchronization operation.

Definition 1: A *data race* occurs when all conditions hold: ① ∧ ② ∧ 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: ⊘① ∨ ⊘② ∨ ⊘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, $a \bigcirc b \wedge ② \wedge ③ \implies ⊘①$.

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

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 5.3). This property ensures that transitive synchronization and sharing of data between threads (see Figure 2(f)) is detected.

4.2 Proof of Correctness

The xDRF analysis (section 3) 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 (①) and they are not synchronization operations (③), the accesses cannot run concurrently (⊘). Formally, in a DRF application: ① ∧ ③ ⟹ ⊘

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 (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 (⊘ ∧ ③ ⟹ 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 (① ∧ ③ ⟹ ⊘).

5. Evaluation of xDRF Regions

This section analyzes the xDRF regions found in applications from the Splash-3 [34] (a modernized, data-race-free version of the Splash-2 [39]) and Parsec-2.1 [7] benchmark suites with standard and simsmall inputs, respectively. We simulate the entire application and collect statistics corresponding to the parallel region of the applications (*region of interest*).

5.1 Static xDRF Regions

We compare the automatic compile-time xDRF delineation with one performed by an expert based on code inspection [33]. Manually delineated xDRF regions act as an oracle of the applications' potential, namely, the largest xDRF regions which could be exposed by perfect analysis, which is equivalent to exposing the maximum number of enclave nDRF regions. We show empirically that no data races occur in practice by employing a consistency checker tool similar to Fast&Furious [32], but extended to support xDRF regions. Figure 5 shows three categories:

- Correctly Non-Enclave: both the compiler and the expert marked the regions as non-enclave (xDRF boundaries). They correspond to barriers, signal-waits, and locks which establish the happens-before order.
- Correctly Enclave: both the compiler and the expert marked the regions as enclave. They show the potential for optimizations.
- Conservatively Non-Enclave: the compiler conservatively marked the region as non-enclave, while the expert detected that the region can be safely enclave.

As one can observe, the compiler misses some optimization opportunities but is always correct. In general, the xDRF analysis performs well and approaches the oracle delineation on *Dedup*, *FFT*, *Fluidanimate*, *LU*, *Radix*, and *Streamcluster*. Other applications, such as *Barnes*, *FMM*, *Radiosity*, *Water-Nsq*, and *Water-Sp* miss optimizations 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. By assuming that each task may 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. *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. Similar to *Barnes*, the compiler does not detect that only one thread actually can execute this code region and safely 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 level of the function, similar to different iterations of a loop).

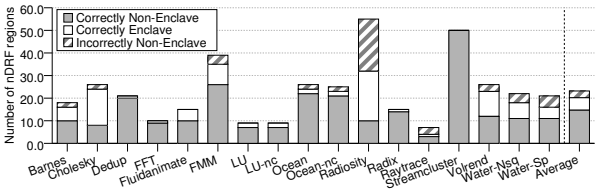


Figure 5. Compiler vs Manual delineation of xDRF regions.

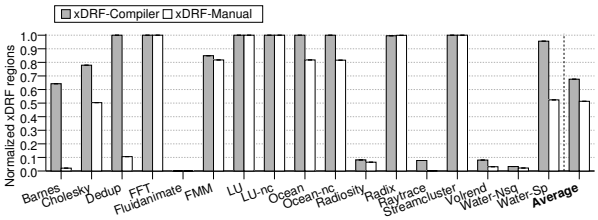


Figure 6. Number of executed xDRF regions.

5.2 xDRF Regions at Runtime

Figure 6 plots the number of dynamic instances of xDRF regions normalized to the number of executed DRF regions (not shown). In this figure lower is better, meaning that less but larger xDRF regions have been found. Figure 6 complements Figure 5 by actually counting, at runtime, which regions are executed more frequently. If the normalized bar equals 1, $\#xDRF = \#DRF$, this indicates that no DRF regions could be merged into the same xDRF region.

The first bar (*xDRF-Compiler*) shows the results of the automatic delineation, while the second bar (*xDRF-Manual*) emphasizes the maximum potential of the applications.

Some applications by their construction do not extend DRF regions (high percentage of Non-Enclave regions), as they synchronize mainly based on barriers and signal-wait constructs, which do not permit extending the xDRF regions across the synchronization points (*FFT*, *LU*, *LU-nc*, *Radix*, and *Streamcluster*). On the other hand, although *Dedup* contains a small percentage of enclave regions, they are executed in a loop, which accounts for a large number of xDRF regions in practice. Applications with high potential (*Radiosity* and *Volrend*), indeed show a low number of large xDRF regions. An interesting observation is that although the compiler is overly conservative in *Radiosity*, the regions conservatively marked as non-enclave are not on the frequently executed path, hence they do not impact the total number of executed

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 ($\times$1 L1)
Directory cache hit time	2 cycle
Memory access time	160 cycles
Topology	Mesh 2D
Flit size, link time	16 bytes, 1 cycle

Table 1. System parameters.

xDRF regions. In contrast, in *Barnes*, *Cholesky*, *Water-Nsq*, and *Water-Sp* a few conservative non-enclave regions executed frequently change the total number of xDRF regions compared to *xDRF-manual*. *Fluidanimate*, *Radiosity* and *Raytrace* show the highest potential, both with automatic and manual delineation, with a few large xDRF regions.

6. Optimizing Cache Coherence Using xDRF

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

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

SPEL++ [30] 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, 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.

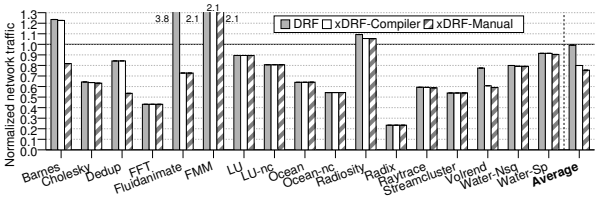


Figure 7. Network traffic normalized to *Directory*.

6.2 Simulation Methodology

We employ the GEMS simulator [26] fed with memory accesses generated by PIN [25] as explained by Monchiero *et al*. [27]. The interconnection network has been modeled with GARNET [5], included in the GEMS toolset. Reported energy consumption has been obtained with McPAT [21], assuming a 32nm process technology.

We evaluate a system with 64 in-order cores and L1 caches kept coherent by a directory protocol. Table 1 shows the parameters of the simulated system. We compare SPEL++ using straightforward DRF delineation (*DRF*), i.e. regions between any two consecutive nDRF regions, manual xDRF (*xDRF-Manual*) delineation, and automatic xDRF delineation (*xDRF-Compiler*) to a directory protocol (*Directory*).

6.3 Performance Results

Larger xDRF regions imply a lower number of region boundaries, which in turn leads to decreasing the number of flush operations required to keep coherence of DRF accesses. This results in less invalidations and write-backs, and thus less cache misses and coherence traffic. Our simulations show that the automatic xDRF delineation can avoid, on average, 34.5% of the cache misses that take place when coherence is enforced at every DRF boundary.

This reduction impact network traffic. Figure 7 shows the network traffic generated by the applications, normalized to *Directory* (not shown). SPEL++ for DRF regions is able to reduce the network traffic in all 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 (see Figure 6). On average, the network traffic is reduced by 20.7%, which approaches the ideal, manual delineation (23.4%).

Figure 8 shows the execution time normalized to a directory protocol (not shown). Again, we observe that, on average, SPEL++ with mere DRF delineation is on-par or slightly outperformed 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

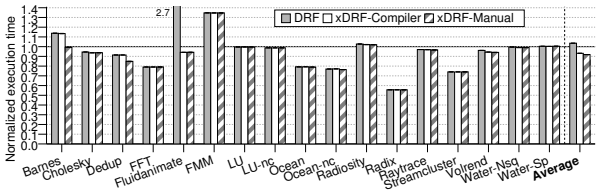


Figure 8. Execution time normalized to *Directory*.

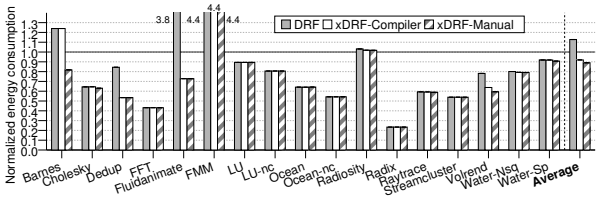


Figure 9. Energy consumption normalized to *Directory*.

(8.1%). The only exception is *Barnes*, where the automatic delineation does not reach the performance of the manual one. This is due to false-positive conflicts reported by the compiler, as it cannot detect that a region of code is only executed by the main thread.

Finally, Figure 9 plots the energy expenditure normalized to *Directory* (not shown). Clear improvements are observed when using the xDRF delimitation compared to both *DRF* and *Directory*. On average, the automatic compile-time delineation saves 11.7% of the energy consumed by a directory protocol, which is approaches the ideal, manual xDRF delineation (16.5%).

7. Related Work

We have proposed methods to identify and exploit xDRF regions in OpenMP applications [29], but previous techniques were not suitable for "unmanaged" (e.g. pthreads) parallel applications based on the fork-join with synchronization model, addressed in this work.

Joisha et al [18] 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. [10, 11] 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 [22, 23, 35] 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. [35] 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 [2, 11, 13–15, 28] to combat races. Static techniques [2, 13, 28] must be conservative and therefore report false-positives, while dynamic techniques [11, 14, 15, 24] miss races which do not occur in the observed execution and introduce high overheads. Acculock [40] 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 [8] 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 [24] relies on hardware support 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.

8. Conclusions

We describe an automated compile-time classification of "unmanaged" parallel programs which delineates DRF regions and identifies extended data-race-free regions (xDRF). xDRF are regions of code which 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.

Acknowledgments

We thank the reviewers for feedback on this work. This work is a result of the internship 19998/IV/15 funded by the Fundación Séneca-Agencia de Ciencia y Tecnología de la Región de Murcia under the "Jiménez de la Espada" program for mobility, cooperation and internationalization. This work was 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] LLVM Alias Analysis. website, Mar. . URL `http://llvm.org/docs/AliasAnalysis.html`.

[2] 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)*, 28(2):207–255, Mar. 2006.

[3] S. V. Adve and H.-J. Boehm. Memory models: A case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, Aug. 2010.

[4] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *17th Int'l Symp. on Computer Architecture (ISCA)*, pages 2–14, June 1990.

[5] 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)*, pages 33–42, Apr. 2009.

[6] 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)*, 60 (4):472–483, Apr. 2011.

[7] 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)*, pages 72–81, Oct. 2008.

[8] 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)*, pages 241–259, Oct. 2015.

[9] 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)*, pages 155–166, Oct. 2011.

[10] 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)*, pages 22–29, June 2011.

[11] 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)*, pages 467–484, Oct. 2012.

[12] 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)*, pages 292–304, Oct. 2015.

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

[14] 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)*, pages 121–133, June 2009.

[15] 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)*, pages 337–348, Mar. 2014.

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

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

[18] 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*, pages 623–636, Jan. 2011.

[19] S. Kaxiras and A. Ros. A new perspective for efficient virtual-cache coherence. In *40th Int'l Symp. on Computer Architecture (ISCA)*, pages 535–547, June 2013.

[20] 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)*, pages 75–88, Mar. 2004.

[21] 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)*, pages 469–480, Dec. 2009.

[22] 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)*, pages 501–512, Sept. 2010.

[23] 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)*, pages 231–240, Sept. 2012.

[24] 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)*, pages 210–221, June 2010.

[25] 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)*, pages 190–200, June 2005.

[26] 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*, 33 (4):92–99, Sept. 2005.

[27] M. Monchiero, J. H. Ahn, A. Falcón, D. Ortega, and P. Faraboschi. How to simulate 1000 cores. *Computer Architecture News*, 37(2):10–19, July 2009.

[28] 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), pages 308–319, June 2006.

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

[30] A. Ros and A. Jimborean. A hybrid static-dynamic classification for dual-consistency cache coherence. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 27(11): 3101–3115, Nov. 2016.

[31] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *21st Int'l Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 241–252, Sept. 2012.

[32] 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)*, pages 1–6, Jan. 2015.

[33] 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)*, pages 433–434, Sept. 2016.

[34] 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)*, pages 101–111, Apr. 2016.

[35] 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)*, pages 524–535, June 2012.

[36] 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)*, pages 160–170, Mar. 2016.

[37] 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)*, pages 545–559, Mar. 2015.

[38] 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)*, pages 13–26, Mar. 2013.

[39] 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)*, pages 24–36, June 1995.

[40] 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)*, pages 201–212, Apr. 2011.