Region-Based May-Happen-in-Parallel Analysis for C Programs

Peng Di, Yulei Sui, Ding Ye and Jingling Xue
School of Computer Science and Engineering, UNSW Australia
{*pengd, ysui, dye, jingling*}*@cse.unsw.edu.au*

***Abstract*—The C programming language continues to play an essential role in the development of system software. May-Happen-in-Parallel (MHP) analysis is the basis of many other analyses and optimisations for concurrent programs. Existing MHP analyses that work well for programming languages such as X10 are often not effective for C (with Pthreads).**

This paper presents a new MHP algorithm for C that operates at the granularity of code regions rather than individual statements in a program. A flow-sensitive Happens-Before (HB) analysis is performed to account for fork-join semantics of Pthreads on an interprocedural thread-sensitive control flow graph representation of a program, enabling the HB relations among its statements to be discovered. All the statements that share the same HB properties are then grouped into one region. As a result, computing the MHP information for all pairs of statements in a program is reduced to one of inferring the HB relations from among its regions.

We have implemented our algorithm in LLVM-3.5.0 and evaluated it using 14 programs from the SPLASH2 and PARSEC benchmark suites. Our preliminary results show that our approach is more precise than two existing MHP analyses yet computationally comparable with the fastest MHP analysis.

***Keywords*-MHP; May-Happen-in-Parallel; concurrent program; multi-threading; static analysis;**

I. Introduction

May-Happen-in-Parallel (MHP) analysis determines statically whether a given pair of statements in a concurrent program may be executed in parallel or not. This analysis serves as a cornerstone of many other static and dynamic analyses on detecting, for example, data races [1], [2] and deadlocks [3]. However, determining precisely whether all pairs of statements in a concurrent program may happen in parallel or not is NP-complete [4]. As a result, most of previous MHP algorithms either achieve precision by taking advantage of some language features in a particular programming language or sacrifice precision by computing approximated results.

The X10-like languages provide structured language features to simplify parallel programming. Their concurrency control constructs, `async` and `finish`, supports async-finish parallelism, by avoiding arbitrary uses of `join`s to make stronger scheduling guarantees.

Unstructured languages such as C/C++ and Java remain mainstream in modern software development. Unlike structured language constructs, unstructured and low-level `fork-join` constructs allow programmers to express rich and complicated patterns of parallelism. However, such flexible non-lexically-scoped parallelism poses a major challenge to scalable and precise MHP analysis. The techniques developed for structured languages [5], [6], [7], [8] cannot be directly applied to unstructured languages. In C programs, for example, a thread may outlive its spawning thread or can be joined partially along one program path (a partial join) or indirectly in one of its child threads (a nested join). Thus, a sophisticated interprocedural analysis is needed to capture the MHP relations in C programs.

Previously, a MHP analysis for Java [9] introduces an abstract thread structure analysis to capture the fork and join interactions among threads. However, this analysis ignores partial and nested joins, which are some limitations to be overcome in this paper.

In this paper, we present a new MHP algorithm to analyse C programs. We perform a flow-sensitive Happens-Before (HB) analysis for a program by considering the fork-join semantics of Pthreads on an interprocedural thread-sensitive control flow graph representation of the program. All the statements that share the same HB properties in the program are grouped into one region, so that computing the MHP information for all pairs of statements in the program is reduced to one of inferring the HB relations from among its regions.

Our contributions are summarised as follows:

- We introduce a new MHP analysis that models the fork-join semantics of Pthreads in a C program by performing an interprocedural flow-sensitive analysis on a thread-sensitive control flow graph representation of the program.
- We present a region partitioning algorithm to build a Region Relation Graph (RRG) for a program in order to compute effectively the MHP information for the program by operating at the granularity of its code regions rather than individual statements.
- We have implemented our algorithm in LLVM-3.5.0 and evaluated it using 14 programs from the SPLASH2 and PARSEC benchmark suites. Our preliminary results show that our approach is more

precise than two existing MHP analyses yet computationally comparable with the fastest MHP analysis.

II. Static Thread Model

A. Abstract Thread

An *abstract thread*[1] refers to a call of *pthread_create()* at a fork site during the analysis. Abstract threads are modelled context-sensitively so that a thread t always refers to a context-sensitive fork site, i.e., a unique runtime thread unless t is multi-forked, in which case, t may represent more than one runtime thread.

Definition 1 (Multi-Forked Threads)**.** A thread t is a *multi-forked thread* if its fork site $f_{t_m \to t}$ resides in a loop, recursion or its spawner thread t_m is multi-forked.

For a non-multi-forked thread, its context-sensitive fork site is cloned for its different calling contexts so that the thread uniquely identifies one single runtime thread.

We use $f_{t_m \to t}$ to denote a *fork site*, where $t_m \to t$ represents a spawning relation such that a *spawner thread* t_m creates a *spawnee thread* t. Similarly, $j_{t_m \to t}$ denotes a *join site*, where $t_m \to t$ represents a joining relation such that a spawnee t is joined by its spawner t_m. For a thread t, we write $\mathcal{S}_t$ to stand for the start procedure of t, where the execution of t begins.

B. Thread-Sensitive Control Flow Graph

Given a program, its interprocedural control flow graph (ICFG [10]) is a directed graph. A node represents a basic block containing a sequence of statements. An intraprocedural edge from one block to another represents the flow of control between the two blocks. All interprocedural edges represent the calling relations across the procedures. Indirect calls can be found using Andersen's pointer analysis [11], [12], [13].

An ICFG can be augmented to yield a thread-sensitive interprocedural control flow graph (TCFG) by adding `fork` and `join` edges to represent their thread spawning and joining relations, respectively, as highlighted by the dashed arrows in Figure 1. For a thread t, we write G_t to represent the subgraph of the TCFG corresponding to t, from the entry of t's start procedure $\mathcal{S}_t$ to the exit of $\mathcal{S}_t$, including all reachable callees of $\mathcal{S}_t$ but excluding the statements in any spawnee thread of t.

C. Modeling Thread Joins

To handle joins effectively, we distinguish direct (or immediate) and indirect (or nested) joins. In particular, indirect joins are recursively defined below.

Definition 2 (Direct and Indirect Joins)**.** A join site $j_{t_m \to t}$ represents a *direct join* of t in t_m. In addition, a join site $j_{t_m \to t'}$ represents an *indirect join* of t, where $t \neq t'$, if every execution path in t' contains a join site $j_{t' \to t}$, which is either a direct or indirect join of t in t'.

[1]The term "thread" means "abstract thread" through the following sections; all runtime threads will be noted explicitly.

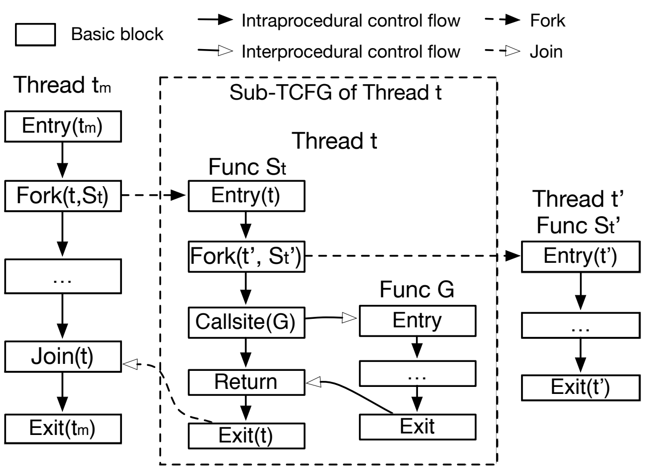


Figure 1: A TCFG and its sub-TCFG for thread t.

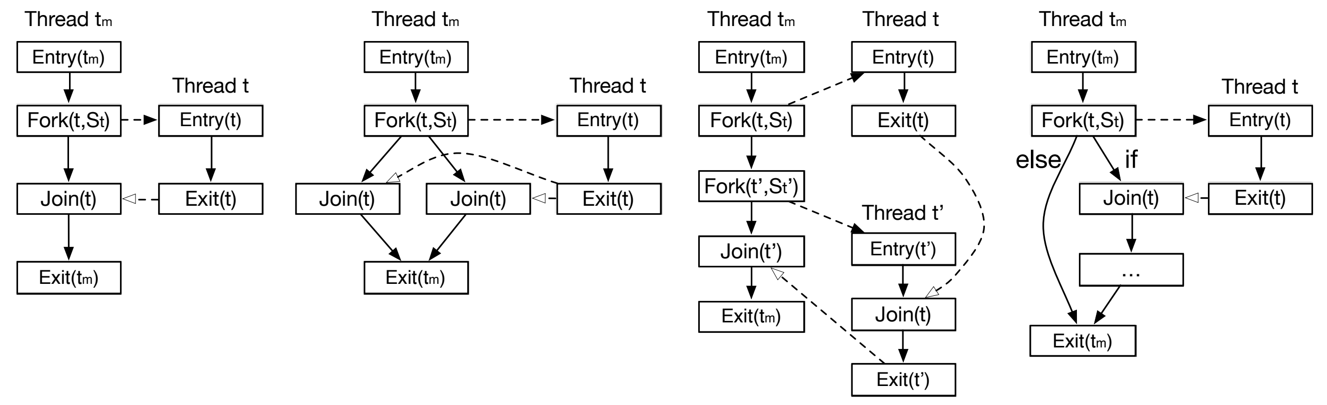
Therefore, a join site $j_{t_m \to t'}$ that represents a join for a thread t is direct if $t = t'$ and indirect otherwise. Figure 2 illustrates four different join patterns to be explained below. Each `Join(`t`)` (`Join(`t'`)`) represents a direct join for t (t'). In addition, as illustrated in Figure 2(c), `Join(`t'`)` represents an indirect join of t in t_m.

In C programs, a thread can be joined fully along all program paths or partially along some but not all paths. Both cases need to be handled precisely.

Definition 3 (Full and Partial Joins)**.** A join site $j_{t_m \to t'}$ that represents a join of a thread t (where t and t' may or may not be the same) is a *full* (*partial*) *join* of t in t_m if it is reachable by its corresponding fork site $f_{t_m \to t}$ along all (some but not all) program paths.

Existing MHP analyses [9], [14] handle a join site conservatively. A join site $j_{t_m \to t}$ is considered only if $j_{t_m \to t}$ post-dominates its corresponding fork site $f_{t_m \to t}$ (as in the case of Figure 2(a)) but ignored otherwise (as in the more complex join cases illustrated in Figures 2(b) – (d)). In Figure 2(b), t is joined fully in t_m at two different join sites in the two branches. In Figure 2(c), t_m creates two threads t and t', with t being joined fully with t_m via a join site in t' in an indirect manner. In Figure 2(d), t can outlive its spawner t_m via a partial join in the `if` branch. If the `if` branch is taken, t terminates after the partial join. If the `else` branch is taken, t may still be alive even after t_m has finished its execution.

Conservatively handling the above join behaviours may lead to imprecise MHP results. Figure 3 shows a partial join from `x264` in the PARSEC benchmark suite, where the thread identified by *h→thread_handle* is joined



(a) Full (post-dominating) join (b) Full (branch-wise) join (c) Full (nested) join (d) Partial join

Figure 2: Full and partial join patterns. In (c), `Join`(t') is not only a direct join of t' but also an indirect join of t.

```
// file encoder/encoder.c
1312 static int x264_slices_write(x264_t *h){
        ......
1327    x264_stack_align(x264_slice_write,h);
1328    i_frame_size =
            h->out.nal[h->out.i_nal -1].i_payload;
        ......
1340 }

1355 int  x264_encoder_encode(...){
        ......
1585    if(h->param.i_threads > 1)
1586    {
1587       pthread_create(&h->thread_handle,
              NULL,(void*)x264_slices_write,h);
1588       h->b_thread_active = 1;
1589    }
        ......
1696 }

1698 static void x264_encoder_frame_end
                                 (x264_t *h,...){
        ......
1705    if(h->b_thread_active)
1706    {
1707       pthread_join(h->thread_handle,NULL);
1708       h->b_thread_active = 0;
           ......
        }
1854 }
```

Figure 3: A partial join in PARSEC.x264.

partially inside an **if** branch in line 1707. Thus, the statements from line 1708 after the join site to the end of the **if** branch do not happen in parallel with any statement in the start procedure **x264_slice_write** of *h→thread_handle*. However, ignoring this join site will result in spurious MHP statement pairs.

Following [9], [15], we presently conduct a may-alias analysis [11], [12], [13] to approximate the set of abstract threads joined at a join site. This can be unsound as a may-analysis is conservative. In general, handling all join sites both soundly and precisely in the presence of multi-forked threads (Definition 1) is difficult. However, it is possible to do so in some common usage scenarios. In Figure 7(a), a "**fork**" loop is first used to spawn a fixed number of threads (lines 18 – 20) and then a "**join**" loop is used later to join all these threads (lines 22 – 24). Such a usage pattern for multi-forked threads can be analysed by taking advantage of LLVM's high-level loop optimiser, Polly, which uses polyhedral abstractions to analyze in-loop memory access patterns. The results computed by Polly can assist our analysis to determine if a multi-forked thread can be joined soundly.

III. Region-Based MHP Analysis

We first perform a flow-sensitive analysis on TCFG to discover the statement-level Happens-Before information by considering the order in which threads execute. We then partition the statements in the program into regions, so that the statements with the same HB properties are grouped together to reduce the overhead incurred in computing all-pair MHP statements.

A. Thread Order Properties

For each context-sensitive abstract thread t spawned at its fork site $f_{t_m \to t}$, we classify the statements (excluding fork and join sites) contained in both G_t and G_{t_m} into five categories to approximate the runtime thread execution order between t and its spawner t_m.

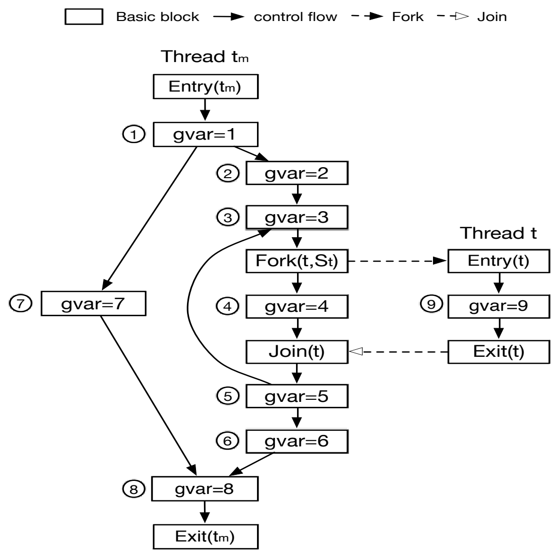
Definition 4 (Thread Order Properties)**.** Let $f_{t_m \to t}$ be a fork site. The statements in G_t and G_{t_m} are classified, by considering direct and indirect joins of t in t_m:

- *BEF* (BEFORE): statements in G_{t_m} that may be executed before the fork site $f_{t_m \to t}$;

```
int gvar; // global variable
void *S(void *x){ gvar = 9; }

void main(){
    pthread_t t;
    gvar = 1;
    if (true){
        gvar = 2;
        for(int  i=0;i<2;i++){
            gvar = 3;
            pthread_create(&t, NULL, S, NULL);
            gvar = 4;
            pthread_join(t, NULL);
            gvar = 5;
        }
        gvar = 6;
    }else{
        gvar = 7;
    }
    gvar = 8;
}
```

(a) Code



(b) TCFG

Figure 4: An example on classifying statements with different thread execution properties. Given t_m as the main thread and t as the multi-forked thread created inside the loop, we have: $P_{t_m\to t}(1) = P_{t_m\to t}(2) = \mathit{BEF}$, $P_{t_m\to t}(6) = P_{t_m\to t}(8) = \mathit{AFT}$, $P_{t_m\to t}(3) = P_{t_m\to t}(4) = P_{t_m\to t}(5) = \mathit{FKJ}$, $P_{t_m\to t}(7) = \mathit{DPP}$, and $P_{t_m\to t}(9) = \mathit{SLA}$.

- *AFT* (AFTER): statements in G_{t_m} that may be executed after a join site $j_{t_m\to t'}$ that represents a (direct or indirect) join of t in t_m (if it exists);
- *FKJ* (FORK-JOIN): statements that appear in some execution path from the fork site $f_{t_m\to t}$ to a corresponding join site $f_{t_m\to t'}$ that represents a join of t (if it exists) or to the exit of G_{t_m} (otherwise);
- *DPP* (DISJOINT PROGRAM PATHS): statements in G_{t_m} that do not appear in any common execution path as the fork site $f_{t_m\to t}$; and
- *SLA* (SLAVE): statements in G_t of the spawnee t.

In what follows, we write $P_{t_m\to t}(s)$ to represent a function mapping a statement s in G_{t_m} and G_t to $\{\mathit{BEF}, \mathit{AFT}, \mathit{FKJ}, \mathit{DPP}, \mathit{SLA}\}$ with respect to one spawning relation $t_m \to t$.

Let us consider an example in Figure 4, where t_m represents the main thread and t a multi-forked thread created inside a loop. Statements ① and ② are annotated with *BEF* since both may be executed before t is created, while statements ⑥ and ⑧ are annotated with *AFT* since both may be executed after the join site. However, statements ③ and ⑤ reside in a loop, causing each to possess both *BEF* and *AFT*, and are thus annotated with *FKJ*. Statement ④ lies on a path between the fork site and join site and is thus labelled with *FKJ*. Statement ⑦ is annotated with *DPP* since the executions of this statement and the fork site are mutually exclusive. Finally, statement ⑨ in G_t is annotated with *SLA*.

B. Flow-Sensitive Static Happens-Before Analysis

In Algorithm 1, we present our flow-sensitive analysis for annotating the statements in a program with their thread order properties. Our algorithm works by considering all thread spawning relations individually (lines 1 and 2). For a spawning relation $t_m \to t$ created at a fork site $f_{t_m\to t}$, we first mark every statement s in G_t as *SLA*: $P_{t_m\to t}(s) = \mathit{SLA}$ (lines 3 and 4). We then mark each statement s in G_{t_m} so that $P_{t_m\to t}(s) \in \{\mathit{BEF}, \mathit{AFT}, \mathit{FKJ}, \mathit{DPP}\}$ by Definition 4 (lines 5 – 24). We do so by first partitioning G_{t_m} into two subgraphs, $G^R_{t_m}$ that contains all statements reachable by the fork site $f_{t_m\to t}$ along some program paths either forwards or backwards and $G^N_{t_m}$ that contains the remaining unreachable statements (line 5). This is achieved by solving a data-flow problem but omitted here.

For every statement s in $G^N_{t_m}$, we set $P_{t_m\to t}(s) = \mathit{DPP}$ (lines 6 and 7). For the statements in $G^R_{t_m}$, we solve a forward data-flow problem $(V, \sqcap, F)$ (lines 8 – 24). We use the semilattice shown in Figure 5, where $V = \{\mathit{BEF}, \mathit{AFT}, \mathit{FKJ}, \top\}$ and $\sqcap$ is the meet operator. Note that if a fork site and its corresponding join site appear in recursion or a loop, a statement can have both *BEF* and *AFT* properties, which will be combined into *FKJ*. F represents the set of transfer functions used for the statements in $G^R_{t_m}$, given in lines 18, 20 and 22. Note that

Algorithm 1: Annotating Thread Order Properties

```
1  $F \leftarrow$ set of all fork statements in the program
2  foreach $\mathrm{f}_{\mathrm{t}_m \to \mathrm{t}} \in F$ do
3      foreach statement $s$ in $\mathrm{G}_\mathrm{t}$ do
4          $P_{t_m \to t}(s) \leftarrow SLA$
5      Partition $G_{t_m}$ into $G^R_{t_m}$ that contains the
       statements reachable by $f_{t_m \to t}$ either forwards or
       backwards along some paths in $G_{t_m}$ and $G^N_{t_m}$
       that contains the remaining statements
6      foreach statement $s$ in $G^N_{t_m}$ do
7          $P_{t_m \to t}(s) \leftarrow DPP$
8      foreach statement $s$ in $G^R_{\mathrm{t}_m}$ do
9          $P_{t_m \to t}(s) \leftarrow \top$
10     $W \leftarrow$ set of statements in $G^R_{t_m}$
11     Let $s_{entry}$ be the entry statement in $G^R_{t_m}$
12     $W = W \cup \{s_{entry}\}$
13     $P_{t_m \to t}(s_{entry}) \leftarrow BEF$
14     while $W \neq \emptyset$ do
15         $s$ = a statement removed from $W$
16         foreach $s' \in succ(s)$ in $G^R_{\mathrm{t}_m}$ do
17             if $s'$ is the fork site $\mathrm{f}_{\mathrm{t}_m \to t}$ then
18                 $P_{t_m \to t}(s') = P_{t_m \to t}(s') \sqcap FKJ$
19             else if $s'$ is a join site $\mathrm{j}_{\mathrm{t}_m \to t'}$
               representing a join of $t$ then
20                 $P_{t_m \to t}(s') = P_{t_m \to t}(s') \sqcap AFT$
21             else
22                 $P_{t_m \to t}(s') = P_{t_m \to t}(s') \sqcap P_{t_m \to t}(s)$
23             if $P_{\mathrm{t}_m \to \mathrm{t}}(s')$ has changed then
24                 $W = W \cup \{s'\}$
```

$succ(s)$ denotes the set of successors of a statement s in $G^R_{t_m}$. To compute $P_{t_m \to t}(s)$ iteratively for each statement s in $G^R_{t_m}$, we first perform the standard initialisation in lines 8 and 9. We then perform a forward data-flow analysis to annotate each statement s in $G^R_{t_m}$ with *BEF*, *AFT* or *FKJ* (lines 10 – 24). Three different transfer functions are applied when processing three kinds of statements: (1) the (unique) fork statement $f_{t_m \to t}$ (lines 17 and 18), (2) its corresponding join statements $j_{t_m \to t'}$ for joining t (if any) (lines 19 and 20), and (3) the remaining statements (lines 21 and 22).

C. Region Relation Graph

After having annotated all the statements in a program with their thread-order properties, we proceed to partition the program into regions so that the statements in the same region share the same thread-order properties. We achieve this by considering each G_{t_m} individually for all spawning relations $t_m \to t$ created in t_m. Two statements s_1 and s_2 in G_{t_m} are grouped in the same region if and only if $P_{t_m \to t}(s_1) = P_{t_m \to t}(s_2)$ for every possible thread t spawned by t_m. Note that $P_{t_m \to t}(s) = P_{t_m \to t'}(s)$ may not necessarily hold when $t \neq t'$. For every region r thus obtained from G_{t_m}, we abuse our notation by writing $P_{t_m \to t}(r)$ to mean $P_{t_m \to t}(s)$ for some $s \in r$ (since $P_{t_m \to t}(s_1) = P_{t_m \to t}(s_2)$ for $s1, s2 \in r$). For every thread t spawned by t_m, let us assume that G_t has been eventually partitioned into n regions (when G_t is processed), $r_1, \ldots, r_n$. By Definition 4, we know that $P_{t_m \to t}(r_1) = \cdots = P_{t_m \to t}(r_n) = SLA$.

$$\begin{tikzcd}
& \top \arrow[dl, no head] \arrow[dr, no head] & \\
BEF \arrow[dr, no head] & & AFT \arrow[dl, no head] \\
& FKJ &
\end{tikzcd}$$

Figure 5: The semilattice for the analysis in Algorithm 1.

$$\texttt{[FORK]} \quad \frac{P_{t_m \to t}(r) = BEF \qquad P_{t_m \to t}(r') = SLA}{r' \longleftarrow r}$$

$$\texttt{[JOIN}^{Full/\boxed{Partial}}\texttt{]} \quad \frac{P_{t_m \to t}(r) = SLA \qquad P_{t_m \to t}(r') = AFT}{r' \longleftarrow r / \boxed{r' \dashleftarrow r}}$$

$$\texttt{[DISJOINT-PATHS]} \quad \frac{P_{t_m \to t}(r) = DPP \quad P_{t_m \to t}(r') = SLA}{r' \longleftarrow r}$$

Figure 6: Rules for constructing the inter-thread edges among the regions in a RRG.

Given a program, its *region relation graph* (RRG) is a directed graph, where each node represents a region and each edge between two regions represents an inter-thread HB relation. We apply the rules given in Figure 6 to construct the inter-thread HB relations for a RRG. We distinguish two types of edges, denoted by $\longleftarrow$ and $\dashleftarrow$, which represent unconditional and conditional HB relations, respectively. `[FORK]` models the HB relations established due to thread creation. `[JOIN]` models the HB relations due to full and partial thread joins. For a full join, $r' \longleftarrow r$ represents the standard HB relation, which happens unconditionally between r and r'. For a partial join, $r' \dashleftarrow r$ represents a conditional HB relation, which happens only conditionally between r and r' when the partial join actually takes place.

Let us now examine the last rule in Figure 6. In `[DISJOINT-PATHS]`, we deal with the case when a region

r in t_m never appears on the same execution path as the fork site $f_{t_m \to t}$ in t_m. We postulate the existence of $r' \longleftarrow r$ (rather than $r \longleftarrow r'$) in order to achieve better precision in practice. This is because for every spawning relation $t_m \to t$, there usually exist a region r in t_m (where $P_{t_m \to t_m}(r) = \mathit{BEF}$) and a region r' in t (where $P_{t_m \to t}(r) = \mathit{SLA}$) such that $r' \longleftarrow r$. This can always happen if the entry node of every sub-TCFG is assumed to contain a no-op, i.e., skip statement.

D. Computing MHP Relations

We are now ready to describe how to compute MHP regions and statements in a program.

Definition 5 (Multi-Forked Regions)**.** A region in G_t is a *multi-forked region* if t is a multi-forked thread.

Let t and t' be two distinct threads. Let r be a region in G_t and r' be a region in $G_{t'}$. We say that r happens before r' and write $r' \Longleftarrow r$ if there exists a directed path from r to r' in the RRG of the program such that the path starts with a sequence of zero or more unconditional (i.e., traditional) HB edges $\longleftarrow$, followed optionally by one conditional HB edge $\dashleftarrow$ at the end.

Two regions r and r' may happen in parallel, denoted $r \# r'$, if *one* of the following two conditions holds: (1) both r and r' are multi-forked regions and (2) neither $r' \Longleftarrow r$ nor $r \Longleftarrow r'$ holds. Recall that if a spawner thread is multi-forked, its spawnee threads will also be multi-forked, but the converse is not true (Definition 1).

Given two regions r_i and r_j, $s_i \in r_i$ and $s_j \in r_j$ may happen in parallel if either (1) $r_i \# r_j$ when $i \neq j$ or (2) r_i is a multi-forked region when $i = j$.

E. Example

Let us apply our MHP analysis to an example in Figure 7. In the code shown in Figure 7(a), there are three spawning relations, $t_m \to t_1$, $t_m \to t_2$ and $t_2 \to t_3$. The main thread t_m creates t_1 in line 14 and joins it partially in line 28. In addition, t_m also forks a multi-forked thread t_2 in line 19 and joins it fully in line 22. The child thread t_2 creates t_3 in line 6 and joins it in line 8. Figure 7(b) depicts the TCFG for the program.

For each spawning relation, we apply Algorithm 1 to annotate program statements with thread-order properties (Figure 7(c)) and partition the statements into regions (Figure 7(d)). Given the partitioned regions, we construct the RRG (Figure 7(e)) according to rules in Figure 6. Finally, we obtain the region pairs that may happen in parallel (Figure 7(f)). All MHP statement pairs can thus be obtained as described in Section III-D.

IV. Evaluation

We have implemented our MHP analysis in LLVM (version 3.5.0). For evaluation purposes, we have selected a set of 14 C programs, including 11 SPLASH2 benchmarks and 3 PARSEC benchmarks, as shown in Table I. All our experiments were conducted on a platform consisting of a 3.00GHz Intel Xeon(R) Quad E5450 processor with 16 GB memory, running Ubuntu Linux (kernel version 3.11.0).

```
1081 for(i=0;i<nprocs-1;i++){ //fork loop
1082   Error=pthread_create(&PThreadTable[i],
                  NULL,(void*)(slave),NULL);
1087 }
     ......

1100 for(i=0;i<nprocs-1;i++){ //join loop
1101   Error=pthread_join(PThreadTable[i],NULL);
1106 }
     ......
```

Figure 8: A multi-forked thread example in `ocean_ncp` from the SPLASH2 benchmark suite.

The source code of each program is compiled into bit code files using clang and then merged together using `LLVM Gold Plugin` at link time stage (LTO) to produce a whole-program bc file. The compiler option `mem2reg` is turned on to promote memory into registers. Andersen's pointer analysis is used to resolve indirect function calls and identify the threads joined at a join site.

To evaluate precision and efficiency of our analysis, we have implemented two previous solutions also in LLVM, one recent MHP analysis, named PCG (Procedural Concurrency Graph [14] and one more precise MHP analysis named as TCT (Thread Creation Tree) [9].

- PCG computes the MHP relations at the granularity of procedures instead of statements. As a result, this analysis is fast but imprecise.
- TCT computes the MHP relations at the granularity of statements based on a thread creation tree. So TCT is more precise but slower than PCG. However, the TCT analysis does not model precisely some complex fork-join behaviours, such as multi-forked threads, partial joins and nested joins.

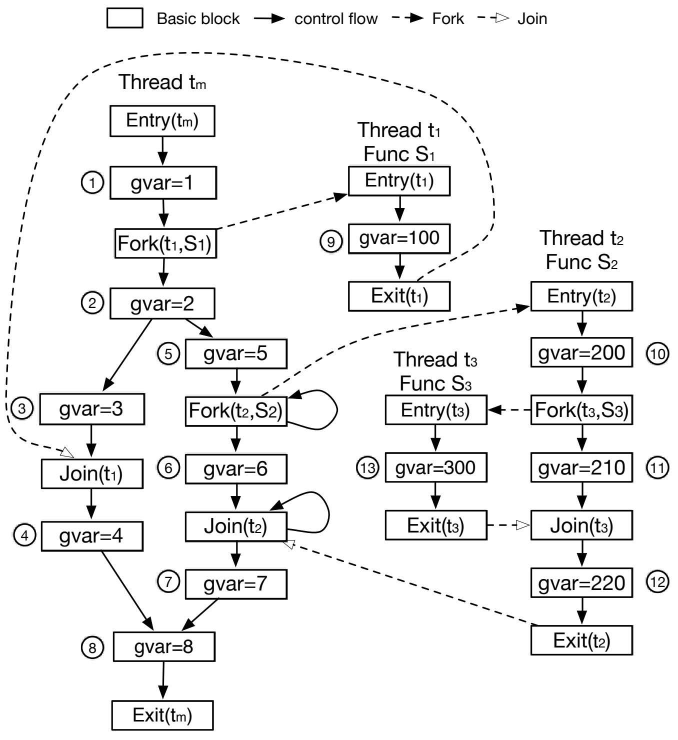
A. Precision

Table I presents the results for PCG, TCT and our MHP analysis referred to as RRG. For each benchmark, we list its code size, the number of memory access statements, i.e., stores/loads, and the number of MHP statements found by each analysis. A statement executed in one thread is classified as a *MHP statement* if it may-happen-in-parallel with at least one statement in another thread. The number of MHP statements for each analysis is obtained by examining all pairs of statements.

Our RRG analysis can identify an average of 30.02% fewer MHP statements than PCG (achieving 48.15% at `raytrace`) and an average of 1.45% fewer MHP statements than TCT (achieving 4.70% at `lu_cb`). RRG

```
pthread_t t1, t2[5], t3;
int gvar; // global variable
void *S3(void *x) {gvar = 300;}
void *S2(void *x) {
    gvar = 200;
    pthread_create(&t3, NULL, S3, NULL);
            //t3:t2's spawnee thread
    gvar = 210;
    pthread_join(t3, NULL);
    gvar = 220;
}
void *S1(void *x) {gvar = 100;}
void main(){ //tm
    gvar = 1;
    pthread_create(&t1, NULL, S1, NULL);
            //t1: partially joined
    gvar = 2;
    if (gvar<4){
        gvar = 5;
        for(int j=0; j<5; j++){
            pthread_create(&t2[j], NULL, S2,
                NULL);//t2: multi-forked
        }
        gvar = 6;
        for(int j=0; j<5; j++){
            pthread_join(t2[j], NULL);
        }
        gvar = 7;
    }else{
        gvar = 3;
        pthread_join(t1, NULL);
        gvar = 4;
    }
    gvar = 8;
}
```

(a) Code



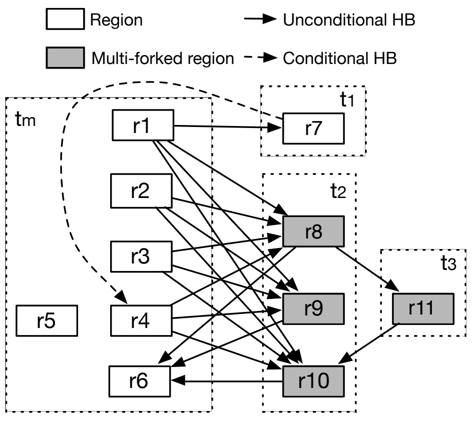
(b) TCFG

Statement \ Relation / Property	$t_m \to t_1$	$t_m \to t_2$	$t_2 \to t_3$
BEF	①	① ② ⑤	⑩
AFT	④	⑦ ⑧	⑫
FKJ	② ③ ⑤ ⑥ ⑦ ⑧	⑥	⑪
DPP	-	③ ④	-
SLA	⑨	⑩ ⑪ ⑫	⑬

(c) Statements with its thread-order properties

Region	Statement	Region	Statement
r_1	①	r_7	⑨
r_2	② ⑤	r_8	⑩
r_3	③	r_9	⑪
r_4	④	r_{10}	⑫
r_5	⑥	r_{11}	⑬
r_6	⑦ ⑧	-	-

(d) Regions



(e) RRG

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	r_9	r_{10}	r_{11}
r_1											
r_2							√				
r_3							√				
r_4											
r_5							√	√	√	√	√
r_6							√				
r_7		√	√		√	√		√	√	√	√
r_8					√		√	√	√	√	√
r_9					√		√	√	√	√	√
r_{10}					√		√	√	√	√	√
r_{11}					√		√	√	√	√	√

(f) MHP statement pairs marked with a "√"

Figure 7: An illustrating example.

Table I: The MHP statements found by PCG, TCT and RRG.

	Benchmark	Application Domain	LOC	Total Number of Statements	Number of MHP Statements		
					PCG	TCT	RRG
SPLASH2	barnes	N-body Simulation	14062	3805	3410	2065	2053
	cholesky	Matrix factorisation	33590	10546	7616	4890	4836
	fft	Complex 1-D FFT	6215	1523	999	712	706
	lu_cb	Matrix triangulation	6063	1586	761	447	426
	lu_ncb	Matrix triangulation	4935	1216	720	676	645
	ocean_ncp	Ocean current simulation	26659	7147	6069	3386	3326
	radiosity	Graphics	25764	7436	4390	2599	2568
	radix	Integer sort	5335	1251	783	665	665
	raytrace	3D rendering	36419	8776	2644	1380	1371
	water_nsquared	Molecular dynamics	10905	3072	2088	1696	1681
	water_spatial	Molecular dynamics	12485	3481	2237	1730	1722
PARSEC	blackscholes	Financial analysis	1077	381	356	312	312
	dedup	Enterprise storage	37931	5317	3741	2841	2830
	x264	Media processing	192981	67929	58080	41231	39960
Total			414421	123466	93894	64630	63101

is slightly more precise than TCT for the set of benchmarks considered since RRG handles some complex fork-join behaviours, such as partial joins as illustrated in Figure 3, more precisely. Figure 8 shows some code snippet from `ocean_ncp` in the SPLASH2 benchmark suite, where a fixed number of threads are forked and joined in two "symmetric" loops. By performing the Polly loop analysis in LLVM, our RRG analysis can identify precisely that any statement in a slave thread does not happen in parallel with the statements after the second loop executed in the main thread. However, TCT handles this scenario conservatively by ignoring the join site in line 1101, resulting in more spurious MHP statements as shown in Table I.

B. *Efficiency*

Figure 9 shows a percentage distribution of RRG's analysis time in each of its three stages: (1) pre-processing (points-to analysis and call graph construction), (2) HP analysis (happens-before analysis and RRG construction) and (3) MHP computation.

To compare RRG with PCG and TCT in terms of efficiency, we examine separately the overheads incurred in building various data structures, e.g., graphs and trees, used by these analyses and the times spent on generating the MHP statements. In the former case, Figure 10 shows that RRG is slightly more efficient than TCT but both RRG and TCT are apparently more costly than PCG. In the latter case, Figure 11 shows that PCG is the fastest (as it is the least imprecise as shown in Table I), and RRG, which operates at the granularity of regions, is 2.1x faster on average than TCT, which operates at the granularity of statements.

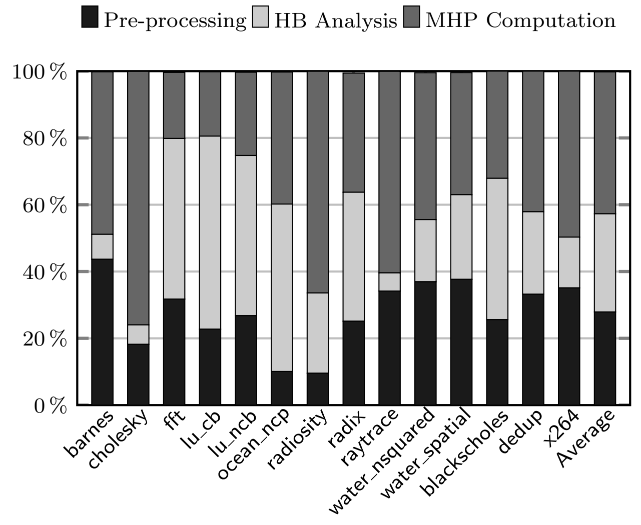


Figure 9: Percentage distribution of RRG's analysis time in its three main analysis phases.

V. Related Work

There has been a lot of studies on MHP analysis. Bristow et al. [16] build an inter-process precedence graph to indicate the synchronisation-imposed execution ordering among processes. Taylor [17] models a concurrency graph based on a reduced flow graph representation of every task. Recently, as parallel platforms become increasingly prevalent, a number of studies have appeared, introducing a variety of advanced techniques

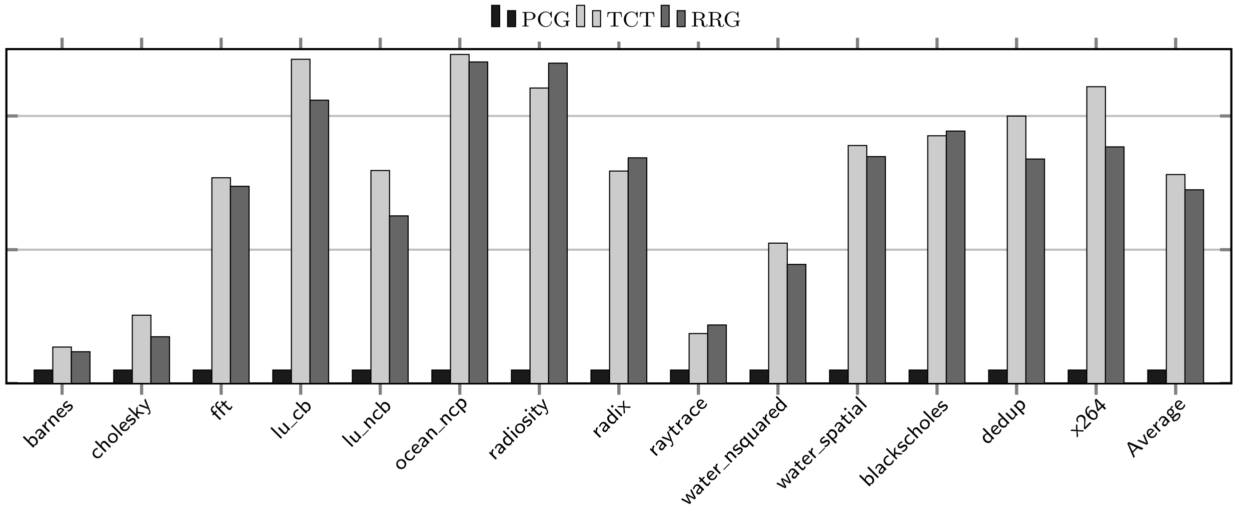


Figure 10: Overheads on building the data structures required (normalised with respect to PCG).

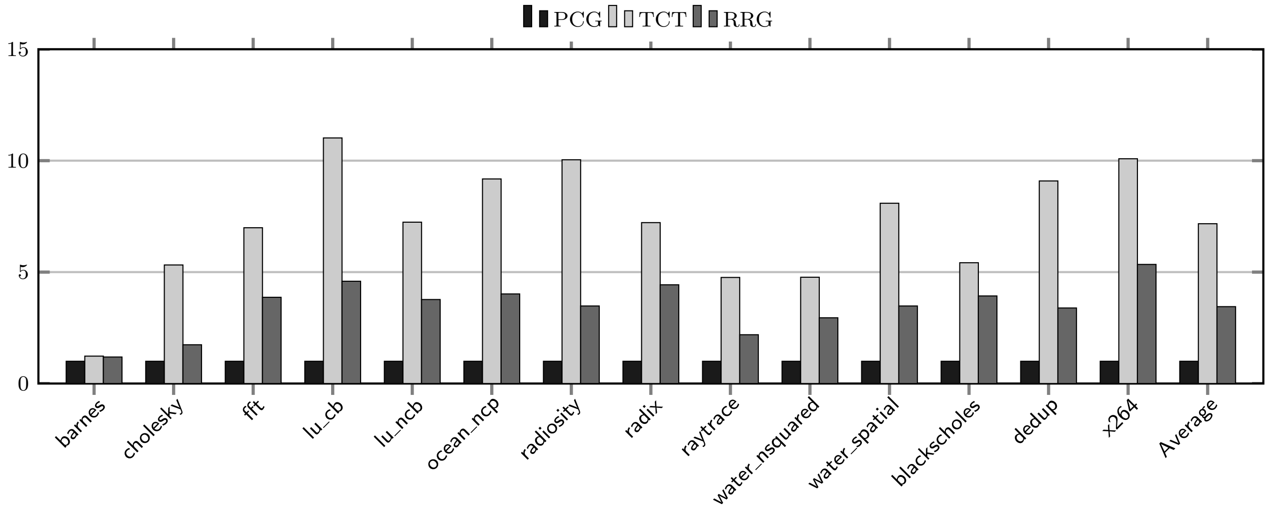


Figure 11: Times spent on generating the MHP statements (normalised with respect to PCG).

for discovering MHP statements in a program.

When it comes to the programming languages with restricted structures, the MHP analysis turns to be more effective due to the simplified problem. Targeting the X10 language, desirable effectiveness can be achieved by an intra-procedural MHP analysis, where simple path traversals are applied in a Program Structure Tree [5]. For improved precision, Lee and Palsberg [7] present a type system that solves a context-sensitive MHP analysis for X10-like structured languages. X10's `async-finish` parallelism model simplifies the inference of escape information, since the `finish` construct ensures that all methods called within its scope terminate before the execution continues to the next instruction. In contrast to these MHP analyses for structured languages, our work focuses on C with Pthreads so that unstructured parallelism is supported. As a result, the presence of more flexible multi-threading patterns renders MHP algorithms for X10-like languages ineffective.

The research for computing MHP information for Java programs is rich in the literature. Based on Parallel Execution Graphs (PEGs), a MHP analysis is applied to Java programs [15], and its scalability was improved later [18]. A drawback with PEGs is that by combining the CFGs of individual threads, a bound on the number of coincident threads modelled is required for the analysis, since the PEGs may potentially grow in size otherwise. To simplify this thread model, Barik [9] describes an efficient MHP algorithm based on a Thread Creation Tree (TCT) that distinguishes threads by their creation sites. However, this analysis is based on the assumption that a parent thread is executed in parallel with each child thread. As a result, many `join` patterns, e.g., nested `join`s, are ignored. In this paper, `fork` and `join` behaviours are more precisely handled. In [19], a flow- and context-sensitive MHP analysis is discussed for the purposes of detecting concurrent errors in Java programs, without considering fully the `join` behaviour.

In the case of C programs (written with Pthreads), the MHP analysis is confronted with substantial challenges. Due to the lack of lexical-scope-based synchronisation and the presence of complex pointer/aliasing relations, both flow- and context-sensitivity are needed. Joisha et al. [14] present a coarse-grained analysis based on Procedural Concurrency Graphs (PCGs) to detect MHP information at the level of procedures. Chen et al. [20] introduced a graph-based MHP algorithm with a context-insensitive thread model. Compared to these, our fine-grained analysis achieves improved precision.

VI. Conclusion

This paper presents a new region-based MHP analysis for C programs. By modelling thread joins (nested and partial joins) more precisely than before and reasoning about happens-before relations at the level of regions (instead of statements), our analysis can achieve better precision than existing MHP analyses while being computationally efficient. As MHP analysis is an important analysis, its improved precision can provide benefits to many client applications, such as program optimisers, bug detectors and security analysers.

VII. Acknowledgements

The authors wish to thank the reviewers for their helpful comments. This work is supported by ARC grants, DP130101970 and DP150102109.

References

[1] X. Xie and J. Xue, “Acculock: Accurate and efficient detection of data races,” in *CGO ’11*, 2011, pp. 201–212.

[2] X. Xie, J. Xue, and J. Zhang, “Acculock: accurate and efficient detection of data races,” *Softw., Pract. Exper.*, vol. 43, no. 5, pp. 543–576, 2013.

[3] M. Naik, C. Park, K. Sen, and D. Gay, “Effective static deadlock detection,” in *ICSE’ 09*, 2009, pp. 386–396.

[4] R. N. Taylor, “Complexity of analyzing the synchronization structure of concurrent programs,” *Acta Informatica*, vol. 19, no. 1, pp. 57–84, 1983.

[5] S. Agarwal, R. Barik, V. Sarkar, and R. K. Shyamasundar, “May-happen-in-parallel analysis of X10 programs,” in *PPoPP ’07*, 2007, p. 183.

[6] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong, “Efficient May Happen in Parallel Analysis for Async-Finish Parallelism,” in *SAS’12*, 2012, pp. 5–23.

[7] J. K. Lee and J. Palsberg, “Featherweight X10: A Core Calculus for Async-Finish Parallelism,” in *PPoPP ’10*, 2010, pp. 25–36.

[8] R. Surendran, R. Raman, S. Chaudhuri, J. Mellor-Crummey, and V. Sarkar, “Test-driven repair of data races in structured parallel programs,” in *PLDI ’14*, 2014, pp. 15–25.

[9] R. Barik, “Efficient computation of May-Happen-in-Parallel information for concurrent Java programs,” in *LCPC’05*, 2005, pp. 152–169.

[10] W. Landi and B. Ryder, “A safe approximate algorithm for interprocedural aliasing,” *PLDI ’92*, vol. 27, no. 7.

[11] L. Andersen, “Program analysis and specialization for the C programming language,” Ph.D. dissertation, 1994.

[12] S. Ye, Y. Sui, and J. Xue, “Region-based selective flow-sensitive pointer analysis,” in *SAS’ 15*, 2014, pp. 319–336.

[13] B. Hardekopf and C. Lin, “Flow-sensitive pointer analysis for millions of lines of code,” in *CGO ’11*, pp. 289–298.

[14] 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 *POPL’11*, 2011, pp. 623–636.

[15] G. Naumovich, “An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs Information for Concurrent Java Programs,” in *ESEC/FSE-7*, 1999, pp. 338–354.

[16] G. Bristow, C. Drey, B. Edwards, and W. Riddle, “Anomaly detection in concurrent programs,” in *ICSE ’79*. IEEE Press, 1979, pp. 265–273.

[17] R. N. Taylor, “A General Purpose Algorithm for Analyzing Concurrent Programs,” *Communications of the ACM*, vol. 26, no. 5, pp. 362–376, 1983.

[18] L. Li and C. Verbrugge, “ A practical MHP information analysis for concurrent Java programs,” in *LCPC’04*, 2004, pp. 194–208.

[19] M. Naik, A. Aiken, and J. Whaley, “Effective static race detection for Java,” in *PLDI’06*, vol. 41, 2006, p. 308.

[20] C. Chen, W. Huo, L. Li, X. Feng, and K. Xing, “Can we make it faster? Efficient may-happen-in-parallel analysis revisited,” in *PDCAT’12*, 2012, pp. 59–64.