Practical MHP Analysis for Java

A. Samuel Moses
Dept of CSE, IIT Madras
Chennai, TN, India
samuelmoses@alumni.iitm.ac.in

V. Krishna Nandivada
Dept of CSE, IIT Madras
Chennai, TN, India
nvk@iitm.ac.in

Abstract

May happen in Parallel (MHP) analysis is one of the most foundational analysis in the context of programs written in parallel languages like Java. The currently known techniques for doing MHP analysis of Java applications suffer from two main challenges: (i) scalability to real-world large applications, (ii) precision in the presence of complex programs. In this manuscript, we address these two issues with a goal of making MHP analysis for Java applications a practical option. We propose a new MHP analysis scheme called GRIP-MHP. It includes techniques to reduce time taken to perform MHP analysis significantly; it does so by using novel schemes to reduce the size of the input graph (representing the original application). GRIP-MHP also addresses many drawbacks in existing techniques, thereby improving the applicability and precision of MHP analysis for real-world Java applications. We implemented GRIP-MHP in the Soot compiler framework. Our experiments show that GRIP-MHP runs successfully in reasonable time on all the tested benchmarks in DaCapo and Renaissance (geomean 20.18× improvement, over prior work). We find that GRIP-MHP performs more precise joining of threads significant leading to improvements in precision of the analysis results: geomean, 1.85× reduction in MHP pairs, over prior work.

CCS Concepts: • **Computing methodologies** → *Parallel programming languages*; • **Software and its engineering** → *Parallel programming languages*; • **Theory of computation** → **Program analysis**.

Keywords: program analysis, parallel/concurrent programs, compressed representation

ACM Reference Format:
A. Samuel Moses and V. Krishna Nandivada. 2026. Practical MHP Analysis for Java. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction (CC '26), January 31 – February 1, 2026, Sydney, NSW, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3771775.3786279

1 Introduction

With multicore systems becoming mainstay, there is a lot of interest in analysing parallel programs. May Happen in Parallel (MHP) Analysis forms a core technique in this space. For each statement in the program, MHP analysis identifies the statements that run in parallel with that statement.

Taylor [17] showed that, in general, for general purpose programming languages, calculating the most precise MHP is undecidable. He also showed that if we assume all control paths of a thread are executable, then the problem is NP-complete. This result has led to development of many conservative MHP analysis techniques.

In the context of Java, there have been three main works in this space. Naumovich et al. [13] proposed one of the first MHP analysis for concurrent Java programs. The algorithm constructs a Program Execution Graph (PEG) which represents the entire control flow of the threads in the program. Li and Verbrugge [12] made some practical modifications to this algorithm to optimize the analysis time and soundness of the algorithm. Their technique also reduced the size of the PEG by removing statements which do not affect the data flow analysis. Barik [3] proposed a non iterative algorithm for a subset of Java programs, with a better time complexity. However, these techniques have three fundamental limitations, making them impractical for real-world use.

(A) Most prior works use a program execution graph (PEG), or a variation thereof, which is a representation of symbolic execution of the input program. Two of the requirements of these techniques are that (i) all the threads are explicitly modelled by cloning, and (ii) the body of each thread should have all the function calls inlined. This leads to large sizes of the PEGs, which in turn significantly increases the analysis time for MHP analysis (as the time complexity is quadratic or higher in terms of the size of the PEG).

(B) Prior works may need the explicit enumeration of all the threads - may not be possible in static analysis. For example, in `h2`, `sunflow` and `xalan` benchmarks of DaCapo [5] threads were created and joined in affine loops; Fig. 1 shows such a sample. Complete enumeration may also not be possible in the presence of recursion.

(C) The handling of thread creation inside loops/recursive functions by these prior works can lead to unsound results, as they may skip necessary MHP pairs, and their imprecise handling of synchronization related constructs may fail to remove pairs of statements that are guaranteed to not run in parallel. Further, these prior techniques do not handle many

```
renderThreads = new SmallBucketThread[scene.getThreads()];
for (int i = 0; i < renderThreads.length; i++) {
   renderThreads[i] = new SmallBucketThread();
   renderThreads[i].start(); }
for (int i = 0; i < renderThreads.length; i++) { ...
     renderThreads[i].join(); ... }
```

Figure 1. Affine loop from DaCapo-Sunflow benchmark

of popular concurrency constructs of Java, such as Thread Pool Executors, Fork Join Tasks, Callables, Futures, and so on that are used in benchmarks like `pmd`, `jme`, `graphchi`, `zxing` from DaCapo [5], and `fj-kmeans` from Renaissance [14].

These restrictions have rendered all of these prior works unusable for most practical Java applications. To the best of our knowledge, there is no known MHP analysis for Java applications, without these limitations. While there have been other interesting prior works [10, 21] on MHP analysis, especially in the context of pthreads/C/C++ languages, their direct applicability/suitability to precisely analyze Java applications remains unclear. This is because Java additionally provides and promotes the use of higher level concurrency primitives. For example, Java provides lexically-scoped exclusion regions through synchronized blocks and allows the use of notify/wait constructs only within synchronized blocks. This presents additional opportunities for improved precision and therefore necessitates the use of a more Java oriented MHP analysis. Hence, the work of Li and Verbrugge [12] (which in turn extends the ideas of Naumovich et al. [13]) is still the most precise MHP analysis technique for Java.

In this manuscript we present two techniques to make the Java MHP analysis practical; we call our overall scheme as GRIP-MHP (Graph Reduction and Improved Precision MHP). We first present techniques to improve the scalability of MHP analysis by reducing the size of the graph being analyzed (without losing any precision). Next, we present extensions to the existing MHP analysis techniques to improve the precision and applicability. We show that GRIP-MHP is the first practical MHP analysis for real-world Java applications. GRIP-MHP can be seen as an extension of the work of Li and Verbrugge [12] to make it practical and scalable to derive precise results for large real-world Java applications.

Though we present GRIP-MHP in the context of Java, we believe that they can be easily applied to MHP analysis of other general-purpose programming languages.

Our contributions:

• We identify the root causes of the impracticality of the existing MHP analysis techniques for Java to be the exploding size of the internal representation of the input programs.
• We present a novel scheme called GRIP-MHP that includes an effective algorithm to greatly reduce the size of concurrent program representations. GRIP-MHP provides a scheme to compute MHP analysis on these reduced representations, and extrapolating the same to the original program.
• GRIP-MHP also extends the precision and coverage of MHP analysis by (i) improving the handling of join and wait statements, (ii) handling of multiple threads created/joined on the same program point, and (iii) sound handling of affine loops.
• We implemented GRIP-MHP in the Soot [19] compiler framework. We show that GRIP-MHP is able to analyze many Java applications, that were otherwise not possible to analyze using existing techniques. We show that compared to prior work, GRIP-MHP leads to significant gains in analysis time (geometric mean of 20.18× improvement), and precision (geometric mean of 1.85× improvement).

2 Background

In this section, we give a brief background of prior techniques by Li and Verbrugge [12] (who in turn extend the work of Naumovich et al. [13]) that will be required in Section 4. Considering space constraints we only focus on those constraints that we need for explanation, or those that we update. Interested reader may get the complete set of constraints from the original manuscripts of Li and Verbrugge [12], Naumovich et al. [13]. Li and Verbrugge define the equations for MHP as follows:

Each node n in the graph is associated with a set $M(n)$ which contains all the statements running in parallel with n, and a set $OUT(n)$ which contains MHP information that needs to be passed to the successors of n. The set $OUT(n)$ is calculated using Equation (1), where $GEN(n)$ is the set of nodes that don't execute in parallel with n, but may execute in parallel with its successors; and $KILL(n)$ is the set of nodes that execute in parallel with n, but will not execute in parallel with its successors.

$$OUT(n) = (M(n) \cup GEN(n)) \setminus KILL(n) \tag{1}$$

The $M(n)$ set is calculated using Equation (2), if n is the first statement in a thread (begin statement).

$$M(n) = M(n) \cup \left(\bigcup_{p \in \text{StartPred}(n)} OUT(p) \right) \setminus N(\text{thread}(n)) \tag{2}$$

If the node is the beginning node of a thread, all nodes belonging to the current thread (represented by $N(thread(n))$) are however removed. Therefore a thread does not run in parallel with itself. However, this rule can cause unsoundness in the results, when the threads may be created in a loop - discussed in Section 4.3.

For a thread-create statement n for creating a thread t, GEN set is defined using Equation (3), where `begin` is the first node of the thread t.

$$GEN(n) = \texttt{begin} \tag{3}$$

When a thread t terminates at a statement `join`, none of the nodes of the thread (represented by $N(t)$) will execute in parallel with the join successors. When a thread enters a `monitor entry` node, the successors of the entry node cannot run in parallel with other nodes inside the same monitor object (represented by $\text{Monitor}_{\text{obj}}$). The *KILL* set captures these cases, as defined in Equation (4)

$$KILL(n) = \begin{cases} N(t), & \text{if } n \text{ is a } \texttt{join} \text{ node} \\ \text{Monitor}_{\text{obj}}, & \text{if } n \text{ is a monitor entry node} \\ \emptyset, & \text{otherwise} \end{cases} \tag{4}$$

The first sub-rule of the above rule for *KILL*, can be applied only when the static analysis can infer with guarantee the thread-object on which the join operation is invoked; we call it as a must-join operation. If must-join cannot be performed, then the last sub-rule is used.

3 Scaling MHP Analysis

Prior works [12, 13] present the MHP analysis on an program representation called PEG (program execution graph). We argue that controlling the size explosion of the PEG, and constructing a small but precise representation of the PEG (called compressed symbolic execution graph; CSEG) can be the key to obtaining precise results in reasonable time. In this section, we present different techniques to efficiently generate CSEG for any given input program.

Li and Verbrugge [12] have shown the importance of precisely identifying different thread objects, synchronization objects and bodies of different threads; the first two tasks related to points-to analysis for thread and synchronization objects, and the last task is related to MHP analysis. Considering the scalability issues in context-sensitive points-to analysis, like many prior works, we also stick to context-insensitive analysis. The prior work also clearly establishes that inlining function calls helps improving precision in these tasks. Our first insight is that both points-to analysis and MHP analyses do not need the same statements that are interesting for each analysis. Consequently, we construct two CSEGs, each containing subsets of nodes of the original PEG:

• **Points-to-stage CSEG**: This execution graph contains statements which are necessary for the MHP analysis, and those which create, and access thread and monitor objects, and also objects which contain thread and monitor object related fields. We obtain more precise points-to information after we run the analysis on this graph.

• **MHP stage CSEG**: This execution graph is a reduced form of the points-to-stage CSEG, wherein only concurrency related statements are kept. Therefore, the points-to information computed on the points-to stage CSEG is applicable to the MHP stage CSEG as well.

While these two CSEGs lead to smaller graphs (compared to the PEGs), in the remaining part of this section, we will present techniques to further reduce these graphs.

3.1 Avoiding Redundant Nodes

Removing statements/nodes which are not necessary for the analysis is an important optimization for analyzing the program in reasonable time. Li and Verbrugge [12] proposed merging into a single node (i) all the nodes of strongly connected components, (ii) sequence of non-branch nodes, if they did not contain any *important* (points-to or MHP analysis related) statements.

However, their scheme persists with a lot of unnecessary branch nodes. For instance, in the example program Figure 2, the sequence of nodes D,E and F,G are merged into one node as shown in Figure 2b. However, since C is not modifying MHP information, it would be advisable to replace all the nodes $A - G$ with a single node.

We next present an efficient algorithm for constructing a compressed symbolic execution graph (CSEG) that only contains statements that are MHP or points-to important. A statement is MHP (or points-to) important if the transfer function of MHP (or points-to) analysis is a non-identity function for that type of statement.

3.2 Basis for new algorithm

It can be seen that the solution of iterative data-flow analysis, associate an *IN* and *OUT* map with each statement (= node in the PEG). For example, as discussed in Section 2, for the MHP algorithms of Li and Verbrugge [12], Naumovich et al. [13] M, and OUT, are the IN and OUT maps.

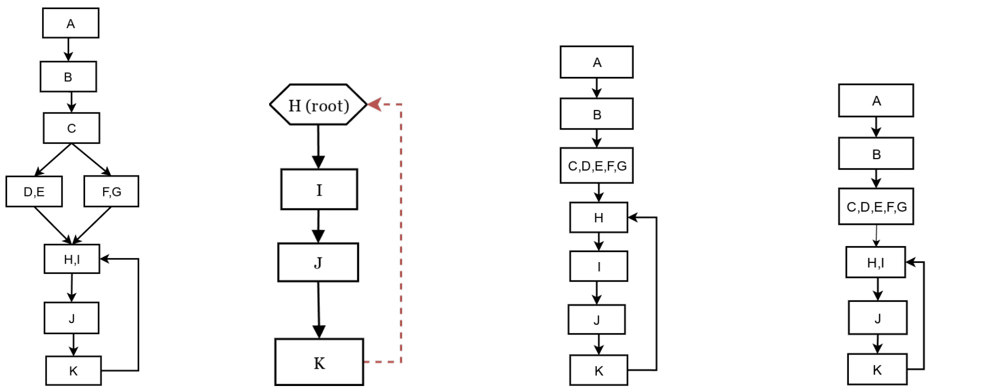
Now we define two sufficient conditions under which the solution of a node s can be represented by the *OUT* solution of its predecessor(s). (i) $IN(s) = \sqcap_{p \in \text{pred}(s)} OUT(p)$, and (ii) s is an *identity-transfer-statement*, that is, the transfer function of s is the identity function. Note I: if each of the predecessor p has the same value of $OUT(p)$ (say x), then $IN(s) = x$. Note II: if s is an identity-transfer-statement then $OUT(s) = IN(s)$. For example, in the MHP analysis of Naumovich et al. [13] we encounter identity-transfer-statements when *GEN* and the *KILL* sets are empty in the definition of the statement.

Similarly the solutions for all the statements of an SCC can be represented by the *OUT* solution of its predecessor(s), if all its statements are identity-transfer-statements, and all inter-SCC predecessors to the nodes of this SCC have same *OUT*.

To identify statements where we do not need to maintain in the CSEG, we compute a novel dataflow map $\mu : L \times \{IN, OUT\} \rightarrow L$, where L is the set of nodes of the SCC-decomposition graph of the PEG. For example, for any node l, $\mu[l][IN] = l'$ (or $\mu[l][OUT] = l'$) implies that the *IN*(or *OUT*) map of node l is same as the *OUT* map of node l'. From this representation of the solution, we can obtain statements which have the same solution and condense them into one representative statement. For any node l, if $\mu[l][OUT] = l$, it will be present in the CSEG; we call such a node as a *leader* node. All nodes whose *IN* and *OUT* solutions match the *OUT*

```
t1:= newthread()  #A
start(t1)         #B
if (cond2) {      #C
  y := y + 1      #D
  x := x + 1      #E
} else {
  y := y - 1      #F
  x := x - 1      #G
}
while (cond3) {   #H
  t2:= newthread() #I
  start(t2)        #J
  join(t2)         #K
}
```

(a) Program pseudo code



(b) Li and Verbrugge [12] Compressed PEG **(c)** Get-Component-SCCs (SCC-3) **(d)** GRIP-MHP PTS stage CSEG **(e)** GRIP-MHP MHP stage CSEG

Figure 2. Graph Inlining Example

solution of the leader are known as *follower* nodes, and will be condensed into a node as a successor of the leader in the CSEG. This method ensures that the obtained MHP results on the CSEG, represents that obtained on the original PEG.

Consider the example code in Figure 2a. It contains eight SCCs, namely seven *SingletonSCCs* : A, B, C, D, E, F, G, and one SCC_{loop} : $\{H, I, J, K\}$. SCC_{loop} cannot be represented by solution of predecessors neither for the MHP stage (since it contains thread-create and join statements that are not identity-transfer function statements), nor for the points-to stage (since it additionally contains a statement which creates a thread creation object). On the other hand the statements of C, D, E, F, G are considered as follower nodes of B, as they are identity-transfer-statements, and their IN and OUT sets match the OUT of B. Therefore, they get condensed as a successor node of B in the CSEG. Figure 2d (CSEG for the points-to stage) and Figure 2e (CSEG for the MHP stage).

3.3 Proposed Algorithm to Compute the Final CSEG

We now present our proposed algorithm to compute the μ map in the process to identify the nodes in the final CSEG. Algorithm 1 shows the driver algorithm for the same.

We go over each procedure (CFG) in the program and then invoke the function `CSEG-Shortlist`, which returns a pair: (S_x, C_{to_proc}), where S_x is the set of statements to be retained, and C_{to_proc} returns the set of SCCs to be further processed. We now explain the working of the functions `CSEG-Shortlist`, and `Get-Component-SCCs`.

For each root (entry) node h `CSEG-Shortlist` (see Algorithm 2) starts by initializing the maps $\mu[IN][h]$ and $\mu[OUT][h]$ to h (line 2) – to indicate that h is a leader node.

It then starts processing the nodes of the SCC-decomposition graph in a topological order. For any SCC C_x, if all inter-SCC predecessors have the same OUT state, and all statements of

Algorithm 1: Driver Program

```
Data: Input Program $\mathbb{P}$
Result: $\mathbb{S}_{selected}$ - set of statements retained in graph
1  $\mathbb{W}_p \leftarrow$ set of graphs representing each procedure in $\mathbb{P}$;
2  $\mathbb{S}_{selected} \leftarrow \{\}$ // Set of selected statements
3  while $\mathbb{W}_p \neq \emptyset$ do
4      $P_x \leftarrow$ select and remove an element from $\mathbb{W}_p$;
5      $(S_x, C_{to\_proc}) \leftarrow$ CSEG-Shortlist($P_x$);
6      $\mathbb{S}_{selected} \leftarrow \mathbb{S}_{selected} \cup S_x$
7      for $c \in C_{to\_proc}$ do
8          $P_{prepared} \leftarrow$ Get-Component-SCCs ($\mathbb{P}, c$);
9          $\mathbb{W}_p \leftarrow \mathbb{W}_p \cup \{P_{prepared}\}$; // Add to worklist
10 return $\mathbb{S}_{selected}$
```

C_x are identity-transfer-statements then the statements of C_x will inherit the OUT map of its inter-SCC predecessors (line 10).

Otherwise, we tentaively mark all the statements of C_x as leaders (line 14). Finally, among these SCCs, those that contain more than one statement, are stored for further processing (line 16),

For each SCC P_x that is kept for further processing, we identify the component SCCs using `Get-Component-SCCs` (Algorithm 3). We collect the entry (root) nodes $\mathbb{P}$ (line 3) and exclude the incoming edges to the entry (root) nodes (line 4). The deleted edges are shown in Figure 2c marked in (dashed) red. Deleting the edges makes the entry nodes with in-degree zero, thereby breaking the SCC. Since all entry nodes are conservatively marked as leaders in the shortlist algorithm, we can safely continue processing.

Algorithm 2: `CSEG-Shortlist`

```
Data: $\mathbb{P}$: Input Procedure
Result: $\mathbb{S}_{selected}$: set of retained statements in the graph,
        $\mathbb{C}_{to\_proc}$: set of SCCs for further processing
($\mathbb{G}$, $\mathbb{C}$) ← ComputeSCCGraph($\mathbb{P}$); // SCC decomposition-graph $\mathbb{G}$, SCCs $\mathbb{C}$
$\mathbb{R}$ ← Roots of $\mathbb{P}$ ;
for $h \in \mathbb{R}$ do                                // For each root node
    $\mu[h][IN] \leftarrow \mu[h][OUT] \leftarrow h$; // Root nodes cannot share state from other nodes
for $C_x \in \mathbb{C}$ in topologically sorted order do
    predSCC ← SCC predecessors of $C_x$;
    if $\forall(a, b) \in Edges(\mathbb{P})$ where $a \in predSCC$ and $b \in C_x$
      have the same $\mu[a][OUT]$
    and All statements in $C_x$ are identity-transfer-statements
      then // All inter-SCC predecessors have the same OUT state, and all statements of $C_x$ are identity-transfer-statements
        Let $a$ be any inter-SCC predecessor of $C_x$;
        for $s \in C_x$ do
            $\mu[s][IN] \leftarrow \mu[s][OUT] \leftarrow \mu[a][OUT]$;
              // same as preds
    else
        for $s \in C_x$ do
            $\mu[s][IN] \leftarrow \mu[s][OUT] \leftarrow s$ ;// can't share from preds
        if $|C_x| = 1$ then Add $C_x$ to $S_{selected}$;
        else Add $C_x$ to $C_{to\_proc}$;
return $\mathbb{S}_{selected}$, $\mathbb{C}_{to\_proc}$;
```

Algorithm 3: `Get-Component-SCCs`

```
Data: $\mathbb{P}$: Input CFG, $\mathbb{C}$: SCC to extract subgraph for
Result: $\mathbb{P}_{prepared}$: Subprogram CFG prepared for
        processing, $\mathbb{R}$: Roots of CFG
Function Get-Component-SCCs($\mathbb{P}$, $\mathbb{C}$)
    $\mathbb{P}_{sub}$ ← Subgraph of $\mathbb{P}$ with nodes and edges of $\mathbb{C}$;
    $\mathbb{R}$ ← statements of $\mathbb{P}_{sub}$ that have external incoming
      edges (from statements $\in (\mathbb{P} - \mathbb{P}_{sub})$)
    $\mathbb{P}_{prepared}$ ← $\mathbb{P}_{sub}$ with incoming edges to $\mathbb{R}$ removed;
    return $\mathbb{P}_{prepared}$
```

3.4 Soundness

Theorem 1. *Assume that, for each node n in the input (SCC-decomposition) graph, CSEG(n) gives the corresponding representative node in the compressed graph. Let M be the MHP map for the input program and M′ be the MHP map for the compressed program. Given any pair of nodes x and y in the input graph,* $y \in M(x) \Leftrightarrow CSEG(y) \in M'(CSEG(x))$.

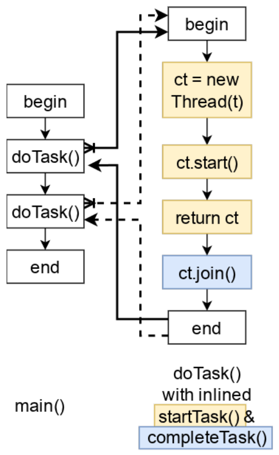
Proof details omitted for space.

3.5 Space Explosion Due to Method Inlining

It is well known that inlining can improve the precision of MHP analysis. However, fully inlining all the methods may not be feasible (due to recursion etc), and may not scale well (due to space explosion). We now describe how we address the space explosion issue by selectively inlining methods; we call this scheme as targeted-inlining.

```
startTask(...){
    ct = new Thread(...);
    ct.start();
    return ct;
}
completeTask(Thread ct){
    ct.join();
}
doTask(...){
   thread = startTask(...);
   completeTask(thread);
}
main(){
   doTask(...);
   doTask(...);
}
```

(a) Program



(b) targeted-Inlined CSEG

Figure 3. Example of targeted-inlining

We use a heuristic which tries to inline functions such that all the concurrency constructs a thread (thread object creation, thread start, join operations and so on) all appear in the same function. All other functions will not be inlined and therefore will be processed in a context-insensitive manner. To handle recursion, we use the scheme of von Praun and Gross [20].

This targeted-inlining enables more precision and helps differentiate runtime instances of thread objects, thereby improving the precision of results in MHP analysis, without excessive bloating of the input graph. As discussed in Section 1, we have observed the existence of thread-factory functions (functions which create, initialize and return a thread object), in the real world applications. Without targeted-inlining strategy defined above, points-to-analysis cannot differentiate between instances of the thread object created at different locations in the program. This also is useful when the thread starts and joins are executed through utility functions like the example in Figure 3.

Note that a limitation of the targeted-inlining method is that shared method bodies introduce unintentional control flow. This can lead to imprecise MHP analysis results. However the result remains sound. Improving the selection criteria for inlining can help mitigate this issue.

In our method we switch to targeted-inlining automatically. We monitor if the fully-inlined graph size reaches a threshold of 100000 units and fallback to targeted-inlined graph in that case.

```
class TA extends Thread{     class TB extends Thread{
    void run(){s1;}}             void run(){
main_function(){                     TA ta = new TA();
    TB mt = new TB();                ta.start();
    mt.start();                      ta.join(); } }
    mt.join();
    s2; }
```

Figure 4. Example illustrating the limitation of the prior works by Li and Verbrugge [12], Naumovich et al. [13]

4 Improving the Precision of MHP Analysis

In this section, we present techniques to improve the precision of prior work on MHP analysis, by handling common programming patterns not handled precisely by the prior work, fixing soundness related issues in the prior work, and handling additional concurrency related constructs.

4.1 Improving Conditions for Must-Joins I

Consider the example (shown in Fig. 4) where, the main thread creates and later joins a thread stored in a variable mt. Assume that this thread also creates and joins another thread ta. When the main thread joins mt, it should ensure that ta is also joined. Otherwise, MHP analysis results will show that the nodes of ta will continue running in parallel with statements in the main thread, after the call to the join mt (for example, s2) – leading to imprecision.

We now propose a scheme to improve the precision of iterative dataflow based MHP algorithms and elaborate this scheme in the framework of Naumovich et al. [13].

4.1.1 New Condition identifying Joining threads. Let $\mathbb{T}_r$ be the set of threads that continue to run in parallel with the joining thread after the join of t_j. Due to the transitive nature of Happens Before constraints like thread joins, any ordering that prevents a statement m of the threads of $\mathbb{T}_r$ to run in parallel with the last statement of t_j, the ordering must be transitively carried over during the join of t_j. Therefore, if a thread t_a is not running in parallel with the last statement of the joined thread t_j, the join statement for can be assumed to be a join for t_a, as well.

We can extend the prior work of Li and Verbrugge [12], Naumovich et al. [13], by considering the following: As per the prior work (see Section 2), for a join statement n of a thread t, $KILL(n) = N(t)$, where $N(t)$ returns all the statements of the thread. For each such n, we update the rule to compute the OUT map.

$$OUT(n) = ((M(n) \cup GEN(n)) \backslash KILL(n)) \cap M(x) \quad (5)$$

where x is the last statement of the thread that is being joined[1]. The terms $(M(n) \cup GEN(n)) \backslash KILL(n)$ (the previous OUT set) and $M(x)$ are monotonic and their intersection is also monotonic. Therefore the updated expression still ensures that the term remains monotonic, and the analysis terminates. OUT being bound by the $M(x)$ helps join all transitively joined threads.

[1]Last statement of every thread is a *thread-end* statement whose transfer function is an identity function.

4.2 Improving Conditions for Must-Joins II

A must-join (see Section 2) at a join-statement can be performed only if the points-to analysis can identify with guarantee the thread instance that is being joined. Previous works use very conservative conditions to ensure that the created and joined thread instances are the same. Given a thread creation statement a, and join statement b, Li and Verbrugge [12] use the following conditions:
• The thread objects of a and b point to the same singleton set of object,
• The allocation site of the thread object of a and b is guaranteed to be executed only once at runtime.

This condition is too restrictive as most real world programs do not guarantee that the allocation site runs only once. There are also cases where the allocation and other thread operations are inside a loop.

We propose a more general approach using the Global Value Numbering (GVN) method[15]. While running the MHP analysis, flow sensitive GVN algorithm is run on the Points-To-Stage CSEG to test if a and b point to the same Global Value Number. If a and b points to the same Global Value Number, then they are guaranteed to hold the same value and therefore point to the same runtime instance of the object. This improvement is made possible due to the relationship between the Points-To and MHP stage CSEGs.

4.3 Threads Running in Parallel with Themselves

Consider the example of threads started inside an unbounded loop at line 5 in Figure 1. Prior works [12, 13] do not discuss techniques to detect the existence of unbounded loops and change the number of abstract threads created at such sites. Therefore the thread start at line 5 will be represented by one symbolic thread t_a. Consider the equation for calculating M set for the begin node of the thread, as shown in Section 2. This definition of the M map makes sure that statements of t_a do not run in parallel with itself. But the threads started in the loop run in parallel with each other, implying that statements of t_a should run in parallel with itself.

Thus, the results of these prior works are sound only when each instance of the thread is explicitly represented by an abstract thread in the PEG, which can be difficult. For instance, when the number of iterations are unbounded or not easy to find statically, this technique of modelling threads becomes impossible. Similarly, if the number of iterations are very large, enumerating all the abstract threads can be very expensive.

To address these issues, we present a general approach to handle threads started in unbounded loops, by fixing the

MHP calculation presented in prior work Naumovich et al. [13] to return the correct MHP solution - even when all threads created at the site are represented by a single abstract thread. We do so by modifying the equation for computing the *OUT* and *M* maps for `begin` nodes:

$$OUT(n) = M(n) \tag{6}$$

$$M(n) = M(n) \cup \left(\bigcup_{p \in \text{StartPred}(n)} \mathrm{M}(p) \right) \tag{7}$$

This ensures that if the start node $(*, start, *)$ is running in parallel with a previous instance of t_a, the nodes of t_a will also run in parallel with itself. However if there is no such statement, t_a will not be running in parallel with itself. This modification ensures that the MHP information is sound and precise.

4.4 Threads Started and Joined in Affine Loops

Starting and joining of array of threads inside affine loops is a commonly observed pattern in concurrent programs. Prior works such as Li and Verbrugge [12], Naumovich et al. [13] simply treated such abstract thread objects as summary objects and hence conservatively ignored the effect of join statements. Works such as Di et al. [10], Zhou et al. [21] mark these threads as multi-fork-join threads and handle them in a more specific manner.

Using the improvements to threads started in loops (Section 4.3), and more precise methods for determining the runtime instance of thread objects (Section 4.2), we present a general condition to handle affine loops that does not require a separate abstract thread model.

We first note that the range of an affine loop can be represented using a three tuple (initVal, finalVal, step), and this can be used to track the runtime instances of array of threads started in a loop, and the runtime instances joined in the loop. For simplicity, we can assume that 'initVal', 'finalVal' and 'step' are all variables in the program. Consider an array of thread-objects `ThArr1` started in an affine loop L1, an array of thread-objects `ThArr2` joined in an affine loop L2. We say that the join operations in L2, will join all the thread started in L1, only if (i) `ThArr1` and `ThArr2` point to the same object instance O_{th} in both the loops – established using the method described in Section 4.2, (ii) there are no writes to the elements of the array object O_{th} between L1 and L2, and (iii) the loops have the same range. To compare two ranges (i_1, f_1, s_1), and (i_2, f_2, s_2), we check that the global value number [15] of i_1, f_1, and s_1, matches that of i_2, f_2, and s_2, respectively.

Only after the completion of the last iteration of L2, can all the started threads be considered to have joined.

4.5 Better MHP for newer concurrency constructs

We now briefly explain how we handle the concurrency constructs beyond thread creations/joining/wait/notify operations.

As per the Java manual [1], for all types of executors, if the `executor.execute()` function is used to start a task, a future is not returned. Therefore it is not possible to mark the end of the task execution. The task maybe considered done only when the executor is shutdown. On the other hand `executor.submit(<Runnable or Callable> b)` returns a future. When `future.get()` is called, the execution is blocked till the task completes. Therefore `future.get()` serves as the join for the task.

The above constructs can be modelled as follows for MHP
• Each `executor.execute()` or `executor.submit()` statement can be seen as equivalent to a `Thread.start()` statement where the body is given by the Runnable or Callable passed as the argument
• When `future = executor.submit()` is used, the `future` object is represents the executing task. This is equivalent to a thread object.
• `future.join()` is equivalent to a thread join.

Handling Fork Join Tasks. A statement of the form `output = executor.invoke(task)` is a blocking call used to execute Fork-Join Tasks. The executor forks multiple threads, where each thread processes some of the tasks. Before executing the successor to the `invoke()` statement, all the threads must complete and join. Thus, this parallelism construct is similar to an affine loop and hence are handled using the scheme discussed earlier in this section.

4.6 Examples

Thread running in parallel with itself. Consider the example code in Figure 1, where threads that are started in an unbounded affine loop. Using the scheme of prior work [12, 13], we will not get sound results, as using Equation (2) all statements of the current thread are removed from the set of statements that may run in parallel with the `begin` statement of the thread. In contrast, using the new transfer function shown in Equation (7) for the `begin` statement, the *M* set of its start predecessor - containing the `begin` statement is added. The MHP solution therefore correctly shows that the `begin` statement runs in parallel with itself.

Further, our improved handling of affine loops (discussed Section 4.4) ensures that in scenarios like the ones shown in Figure 1 all the threads are joined at the end of the second loop. Thereby improving the MHP results involving statements after the second loop. We have found such cases in two of the benchmarks we have studied (h2, sunflow).

Transitive joins. Consider the example code in Figure 4 where Thread `TB` starts and joins Thread `TA`. At the join statement for Thread `TA`, based on the rule for `KILL` in Equation (4), all statements of thread `TA` are added to the `KILL` set.

Therefore no statement of thread TA will be present in the M set of the last statement of Thread TB. When the join statement for TB is processed in the main thread, the OUT set is calculated according to the improved rule for join statement shown in Equation (5). Due to the intersection term with the M set of the last statement of the joining thread TB, the OUT set of the join statement will not contain any statements of thread TA. Thus, our proposed new rule achieves the effect of transitive join of thread TA in the main thread.

5 Implementation and Evaluation

We implemented our proposed GRIP-MHP analysis in the Soot [19] Java framework. It spanned around 8000 lines of code. To evaluate the impact of GRIP-MHP, we ran it on ten popular benchmarks drawn from two suites: DaCapo Chopin [4] and Renaissance [14]. We excluded 13 benchmarks from DaCapo and Renaissance due to Soot's limited Scala support, call graph generation failures, or a lack of concurrency constructs. We also used three hand written programs which cover cases that are otherwise not covered by the benchmarks.

We use Tamiflex [6] to handle reflection in benchmarks. Since previous works do not handle exceptions, we also ignore them in our experiments. Table 1 shows different static properties of the benchmarks we used for the experiments.

Table 1 lists the number of join statements encountered in each of the benchmarks. In four of the benchmarks our technique fell back to targeted-inlining, due to crossing of the threshold (see Section 3.5). In Tables 3 and 5, these benchmarks carry an additional annotation (TI) to indicate the use of the targeted-inlined graph for results. The last column lists some interesting remarks about the benchmarks.

To present an evaluation, we compared our results with that of Li and Verbrugge [12], which can handle a reasonable subset of Java constructs including synchronized blocks and notify-wait. Note that as discussed in Section 1, none of the other prior techniques meet these criteria. To make the work of Li and Verbrugge handle real-world large applications, we extended the original implementation of Li and Verbrugge in Soot to handle method calls that cannot be inlined, using the techniques described by von Praun and Gross [20]; hereafter, we refer to this extension as LV, and use it as our baseline.

To evaluate our proposed approach we answer six research questions, spanning over four dimensions: (A) Analysis time: (RQ1) How fast is GRIP-MHP compared to LV? (RQ2) What is the overhead due to the graph compression technique? (B) Precision of Results: (RQ3) What is the improvement in precision of GRIP-MHP compared to LV, in terms of MHP pairs? (RQ4) How effective is GRIP-MHP in reasoning about must-join conditions? (C) Impact of the efficient representation: (RQ5) What is the impact of the proposed graph compression technique? (RQ6) What is the impact of the proposed targeted-inlining scheme?

Benchmark	NJ	FI	Remarks
DaCapo			
graphchi	1	✓	Uses Callables and Executor
h2	2	✓	Threads are started and joined in affine loops
jme	1	✓	Task is submitted to Executor and waits on Future for result
luindex	-	✗	No join call found.
lusearch	-	✗	No join call found.
pmd	1	✓	Tasks are executed, futures stored in Java collections
sunflow	3	✗	Threads are started and joined in affine loops
xalan	1	✓	Threads are created, tasks are executed in affine loops
zxing	1	✗	Tasks are executed, futures stored in Java collection
renaissance			
fj-kmeans	2	✓	Uses Fork Join Tasks
example programs			
indirect thread-join	6	✓	Contains 3 indirect thread joins
dining-philosophers	5	✓	Uses Thread Factory function
executor-future	1	✓	Tasks are executed, futures are awaited in affine loops

Table 1. Static Properties of Benchmarks; abbreviation: [NI]: Number of joins; [FI:] Ability to **F**ully **I**inline

5.1 Analysis Time

(RQ1) How fast is GRIP-MHP compared to LV?. Table 3 presents the total time (in seconds) taken by GRIP-MHP and LV, along with a breakdown in different stages of each method. For reference, the table also shows the time taken by the in-built SPARK points-to analysis of Soot. The table (column k = column e / column j) shows that GRIP-MHP results in significant speedups compared to LV, up to 526× (geomean 20.18×). We don't show the gains for luindex, as LV ran out of memory and terminated abruptly.

Recall that the four important stages of GRIP-MHP are (i) creation of inlined CSEG for Points to Analysis (see Section 3.3), (ii) Flow insensitive points to analysis run on the inlined CSEG, (iii) A second round of graph compression run on the inlined CSEG for the MHP analysis (iv) MHP analysis on the final CSEG. Comparing the different stages of GRIP-MHP and LV, while GRIP-MHP spends slightly more time in CSEG construction, the gains coming from the MHP analysis stage more than offsets the graph construction times. We see that the actual gains depend on the amount of parallelism in the application, and the number of nodes in the CSEG; these can be seen in the large gains in lusearch, sunflow, and zxing. An important point to note is that the time gains in MHP analysis time are more impressive, as GRIP-MHP realizes the gains while being more precise (see the discussion in RQ3).

(RQ2) What is the overhead due to the graph compression technique? Table 3 shows the time taken to inline and compress graph using techniques of LV (Column c), and using GRIP-MHP (Columns f, g, and h). It can be seen that overhead of GRIP-MHP (sum of columns f, g, and h) is very small, and is comparable to that of LV. This shows that our techniques are able to reduce large graphs quickly. The additional time required for this pass is offset by a significant improvement in the runtime of the MHP analysis.

5.2 Precision Results

(RQ3) What is the improvement in precision of GRIP-MHP compared to LV, in terms of MHP pairs? To compare the precision of GRIP-MHP and LV, we counted the number of MHP pairs (like prior works). Table 5 shows the number of MHP pairs as computed by both the techniques. It can be seen that in benchmarks `fj-kmeans`, `jme`, `h2`, and `sunflow` there is a significant reduction in the number of spurious MHP pairs. The spurious pairs are due to the joined thread previously running in parallel beyond the join statements. We observe a reduction in number of MHP pairs by a geomean 1.85× (up to 7.4×). This directly correlates to the improved handling of must-join conditions by GRIP-MHP (see RQ4). Further, comparing the MHP analysis times in Table 3 (columns i and l), we observe that improved joins also enable GRIP-MHP to converge faster in the MHP analysis.

(RQ4) How effective is GRIP-MHP in reasoning about must-join conditions? Table 4 shows the number of thread joins detected by GRIP-MHP(column a) and the number of thread joins detected by LV (column c). It can be seen that while LV is unable to detect most of the available thread joins, GRIP-MHP is able detect several thread joins. Further, GRIP-MHP is also able to identify joins inside affine loops. We manually studied the benchmarks and found that all sites of thread creation and join are run multiple times. Consequently, LV could not join any thread objects.

Column b of Table 4 shows our estimated joins (based on manual study) in these benchmarks. It shows that GRIP-MHP is able to identify most of the joins accurately. We found in pmd, xalan and zxing, our analysis could not precisely match the join statement with the thread creation statements. We believe that GRIP-MHP can be extended with a more complex analysis in Soot to calculate the equivalences of contents in Java Collections, and the equivalence of integer fields at different points in the program. We leave it as a future work.

5.3 Impact of the Efficient Representation

(RQ5) What is the impact of the proposed graph compression technique? Table 2 shows sizes of different CSEGs to show the effect of our proposed graph compression techniques: For reference we provide in column b sizes of the input in the original input program (JU: Jimple Units) and in column c sizes of the CSEG of LV, with naive inlinling (NI), Comparing for targeted-inlining method, column g containing sizes of MHP-analysis-stage CSEG of GRIP-MHP (MTI) vs column d containing sizes of the MHP-analysis-stage CSEG of LV (MLVTI) we observe that our compression technique reduces the number of nodes by a geomean of 6.99× (up to 14.17×). Comparing for full inlining method, column i containing sizes of MHP-analysis-stage CSEG of GRIP-MHP (MFI) vs column e containing sizes of the MHP-analysis-stage CSEG of LV (MLVFI) we observe that our compression technique reduces the number of nodes by a geomean of 7.5× (up to 22.46×). The impact of the reduced CSEG sizes can be easily seen in the gains in the analysis time as reported in Table 3; MHP Analysis being a flow sensitive analysis with higher time complexity, is sensitive to the graph size. We also report the impact of the graph compression on the analysis times of LV (in column k, Table 3). It clearly shows that compared to column d, the improvements in analysis time is significant. This attests to the generality of our proposed techniques.

(RQ6) What is the impact of the proposed targeted-inlining scheme? Table 2 also shows sizes of different CSEGs to show the effect of targeted-inlining, by showing the graph sizes in the context of full-inlinig: in column e sizes of the CSEG of LV, using the graph compression techniques discussed in the current paper but with full inlining (MLVFI), in column f sizes of points-to-analysis-stage graph of GRIP-MHP using full inlining (PFI), in column h sizes of MHP-analysis-stage graph of GRIP-MHP using full inlining (MFI). Table 2 shows that our proposed graph compression techniques with targeted-inlining (MLVTI) leads to significant reduction in graph sizes even for LV. The table shows that targeted-inlining leads to significant reduction in graph size and its impact is especially visible for large programs such as luindex, lusearch, sunflow and zxing, which otherwise cannot be analyzed.

An interesting point to see is the following: Table 4 also shows that we are able to detect and join all the available joins in the sunflow benchmark despite being only targeted-inlined. This selective inlining preserves the join semantics of large program while enabling a manageable CSEG size.

6 Related Works

MHP analysis has been studied for a long time. For example, [7] used interprocedural precedence graph for anomaly detection in concurrent programs. [17] proved that for general-purpose programming languages without any constraints, precise MHP is undecidable and NP-complete. [18] describes an algorithm to calculate MHP in Ada.

There have been many prior works [11], [2], [16] that describe MHP analysis on X10 [8]. General-purpose languages like Java which allow low-level concurrency constructs are challenging to analyze compared to task-parallel languages like X10 due to lack of a well defined structure.

			LV		GRIP-MHP			
(a) Benchmark	**(b) Sum JU**	**(c) NI**	**(d) MLVTI**	**(e) MLVFI**	**(f) PTI**	**(g) MTI**	**(h) PFI**	**(i) MFI**
fj-kmeans	449	465	165	167	201	24	203	23
graphchi	3721	8312	1273	2874	1493	279	2583	392
h2	1313	1426	452	529	278	64	345	82
jme	1704	1466	559	562	367	61	373	61
luindex	23785	>100000	28986	>100000	43927	6807	>100000	-
lusearch	6294	>100000	3096	>100000	2440	422	>100000	-
pmd	3050	4227	454	584	190	41	240	26
sunflow	17575	>100000	21759	>100000	24068	1839	>100000	-
xalan	780	910	356	368	274	150	285	147
zxing	8053	>100000	3614	>100000	2086	255	>100000	-

Table 2. Number of nodes in CSEG. Details of columns: `(a) Sum JU`: Sum of **J**imple **U**nits (Java program representation used by Soot) of functions that need to be inlined; `(b) NI`: Baseline **N**aive **I**nlining - where all functions are inlined and unimportant statements are not removed; `(c) MLVTI`: **M**HP stage CSEG created using graph compression technique from **LV** and **T**argetted **I**nlining from Section 3.5; `(d) MLVFI`: **M**HP stage CSEG created using graph compression technique from **LV** and **F**ull **I**nlining; `(e) PTI`: **P**oints-To-Analysis stage CSEG of GRIP-MHP, created using graph compression technique from Section 3.3 and **T**argetted **I**nlining Section 3.5; `(f) MTI`: **M**HP stage CSEG of GRIP-MHP, created using graph compression technique from Section 3.3 and **T**artgetted **I**nlining from Section 3.5; `(g) PFI`: **P**oints-To-Analysis stage CSEG of GRIP-MHP, created using graph compression technique from Section 3.3 and **F**ull **I**nlining; `(h) MFI`: **M**HP stage CSEG of GRIP-MHP, created using graph compression technique from Section 3.3 and **F**ull **I**nlining.

		LV			GRIP-MHP						
(a) Benchmark	**(b) Spark**	**(c) MHP CSEG**	**(d) MHP Anly**	**(e) Total**	**(f) PTS CSEG**	**(g) PTS Anly**	**(h) MHP CSEG**	**(i) MHP Anly**	**(j) Total**	**(k) Time Impr.**	**(l) LV-Sim**
fj-kmeans	1.48	0.14	0.05	0.18	0.14	0.01	0.01	0.02	0.18	1×	0.02
graphchi	2.35	0.66	102.77	103.43	0.7	0.02	1.43	2.45	4.59	22.53×	2.51
h2	1.91	0.27	1.26	1.52	0.28	0.01	0.07	0.08	0.44	3.48×	0.09
jme	2.89	0.35	8.4	8.75	0.35	0.01	0.05	0.07	0.48	18.2×	0.07
luindex (TI)	3.2	2.05	>7200	>7200	3.54	0.66	14.04	916.34	934.58	>7.70×	925.33
lusearch (TI)	2.66	0.71	439.04	439.74	0.73	0.03	0.2	1.79	2.75	159.66×	1.79
pmd	2.15	0.39	0.49	0.88	0.35	0.01	0.03	0.03	0.41	2.13×	0.03
sunflow (TI)	2.76	2.52	>7200	>7200	3.23	0.09	4.36	5.99	13.67	>526×	39.8
xalan	1.53	0.2	0.55	0.74	0.21	0.01	0.09	0.2	0.5	1.48×	0.19
zxing (TI)	2.2	0.88	225.11	225.99	0.86	0.01	0.1	0.21	1.19	190.55×	0.17

Table 3. Benchmarks result in **seconds**. Contains the results of Spark analysis and that of Li and Verbrugge [12] for comparison. Abbreviations: `LV-Sim`: MHP algorithm from LV is run on more optimized MHP CSEG of GRIP-MHP

Naumovich [13] showed MHP analysis on Java Programs using data-flow analysis. Program Execution Graph (PEG) was used to model abstract threads in concurrent programs. Lin and Verbrugge [12] improved scalability of [13] by introducing techniques to reduce the size of PEG. Barik [3] modelled abstract threads as a Thread Creation Tree (TCT) based on thread starts and joins. Chen et al. [9] improves the node processing order of iterative data-flow analysis and made the analysis faster than non-iterative data-flow analysis methods. The structure of the tree was analyzed to provide a conservative MHP. Di et al. [10], Zhou et al. [21] present their works in the context of C/pthreads. Di et al. [10] runs a flow-sensitive happens-before analysis to partitions the graph into multiple regions based concurrency properties. This reduces the effective graph size and therefore solves MHP analysis faster. It uses a post-dominator analysis to identify transitive join locations. However it does not solve for the general case. In comparison to all these works, we propose schemes to make the MHP analysis for Java more scalable and precise. Prior works [10, 21] provide a conservative solution for affine loops/ multi-forked threads. Zhou et al. [21] encode concurrency information in static vector clocks instead of

Benchmark	Must-joins			Remarks
	MJ	EJ	LJ	
fj-kmeans	2	2	0	handles fork/join tasks
graphchi	0	0	0	FutureTask object cannot be analyzed well in *Soot*
h2	2	2	0	handles affine loops
jme	1	1	0	executors and futures
luindex	0	0	0	no join calls found
lusearch	0	0	0	no join calls found
pmd	0	1	0	tasks stored in collections
sunflow	3	3	0	handles affine loops.
xalan	0	1	0	affine loop range defined by object’s integer field
zxing	0	1	0	tasks stored in collections
indirect thread-join	6	6	3	GRIP-MHP handles indirect thread joins
dining-philosphers	5	5	0	Points-to on inlined CSEG improves precision
executor-futures	1	1	0	GRIP-MHP handles executors and futures

Table 4. Number of must-joins identified. Abbreviations: `MJ` Must Joins identified in GRIP-MHP, `EJ` **E**stimated **J**oins identified, `LJ` must joins observed by LV.

Benchmark	LV-sim	GRIP-MHP	Improvement
fj-kmeans	84	27	3.11×
graphchi	24208	24208	1×
h2	1320	283	4.66×
jme	1680	368	4.57×
luindex (TI)	15869268	15869268	1×
lusearch (TI)	87492	87492	1×
pmd	46	46	1×
sunflow (TI)	714046	96486	7.4×
xalan	2700	2700	1×
zxing (TI)	7546	7546	1×

Table 5. Number of MHP pairs. `LV-sim`: LV is run on MHP graph of GRIP-MHP

MHP pairs and run a flow-sensitive happens-before analysis. But due to the conservative modeling of the thread creation objects, these works do not handle thread-join precisely. In contrast, we handle threads created/joined in affine loops precisely.

7 Conclusion

In this manuscript, we propose a new MHP analysis scheme called GRIP-MHP. It includes techniques to reduce time taken for performing MHP analysis significantly; it does so by using novel schemes to reduce the size of the input graph (representing the original application). GRIP-MHP also addresses many drawbacks in existing techniques, thereby improving the applicability and precision of MHP analysis for real-world Java applications. We show that GRIP-MHP leads to significant improvements in terms of analysis time (geomean, 20.18×), and precision (geomean, 1.85× reduction in MHP pairs), compared to prior work.

Acknowledgments

This work is partially supported by the SERB CRG grant (sanction number CRG/2022/006971).

References

[1] 2025. https://docs.oracle.com/javase/8/docs/api/java/lang/ref/Reference.html

[2] Shivali Agarwal, Rajkishore Barik, Vivek Sarkar, and Rudrapatna K. Shyamasundar. 2007. May-happen-in-parallel analysis of X10 programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP ’07)*. Association for Computing Machinery, New York, NY, USA, 183–193. doi:10.1145/1229428.1229471

[3] Rajkishore Barik. 2006. Efficient Computation of May-Happen-in-Parallel Information for Concurrent Java Programs. In *Languages and Compilers for Parallel Computing*, Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 152–169.

[4] Stephen M Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*. ACM. doi:10.1145/3669940.3707217

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications* (Portland, OR, USA). ACM Press, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE ’11)*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

[7] G. Bristow, C. Drey, B. Edwards, and W. Riddle. 1979. Anomaly detection in concurrent programs. In *Proceedings of the 4th International Conference on Software Engineering* (Munich, Germany) *(ICSE ’79)*. IEEE Press, 265–273.

[8] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA ’05)*. Association for Computing Machinery, New York, NY, USA, 519–538. doi:10.1145/1094811.1094852

[9] Congming Chen, Wei Huo, Lung Li, Xiaobing Feng, and Kai Xing. 2012. Can We Make It Faster? Efficient May-Happen-in-Parallel Analysis Revisited. In *2012 13th International Conference on Parallel and Distributed Computing, Applications and Technologies*. 59–64. doi:10.1109/PDCAT.2012.59

[10] Peng Di, Yulei Sui, Ding Ye, and Jingling Xue. 2015. Region-Based May-Happen-in-Parallel Analysis for C Programs. In *2015 44th International Conference on Parallel Processing*. 889–898. doi:10.1109/ICPP.2015.98

[11] Jonathan K. Lee and Jens Palsberg. 2010. Featherweight X10: a core calculus for async-finish parallelism. *SIGPLAN Not.* 45, 5 (Jan. 2010), 25–36. doi:10.1145/1837853.1693459

[12] Lin Li and Clark Verbrugge. 2005. A Practical MHP Information Analysis for Concurrent Java Programs. In *Languages and Compilers for High Performance Computing*, Rudolf Eigenmann, Zhiyuan Li, and Samuel P. Midkiff (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 194–208.

[13] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. 1999. An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs. *SIGSOFT Softw. Eng. Notes* 24, 6 (oct 1999), 338–354. doi:10.1145/318774.319252

[14] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: A Modern Benchmark Suite for Parallel Applications on the JVM. In *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity* (Athens, Greece) *(SPLASH Companion 2019)*. Association for Computing Machinery, New York, NY, USA, 11–12. doi:10.1145/3359061.3362778

[15] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *ACM-SIGACT Symposium on Principles of Programming Languages*. https://api.semanticscholar.org/CorpusID:9876007

[16] Aravind Sankar, Soham Chakraborty, and V. Krishna Nandivada. 2016. Improved MHP Analysis. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 207–217. doi:10.1145/2892208.2897144

[17] RichardN. Taylor. 1983. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica* 19, 1 (April 1983). doi:10.1007/bf00263928

[18] Richard N. Taylor. 1983. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM* 26, 5 (May 1983), 361–376. doi:10.1145/69586.69587

[19] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research* (Mississauga, Ontario, Canada) *(CASCON '99)*. IBM Press, 13.

[20] Christoph von Praun and Thomas R. Gross. 2003. Static Conflict Analysis for Multi-Threaded Object-Oriented Programs. *SIGPLAN Not.* 38, 5 (may 2003), 115–128. doi:10.1145/780822.781145

[21] Qing Zhou, Lian Li, Lei Wang, Jingling Xue, and Xiaobing Feng. 2018. May-happen-in-parallel analysis with static vector clocks. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 228–240. doi:10.1145/3168813

A Artifact Appendix

A.1 Abstract

This artifact enables evaluators to reproduce the main experimental results from **Practical MHP Analysis for Java**. It bundles the pre-built bytecode for MHP analysis, benchmarks used in the evaluation and the Docker environments to build and run the artifact. Automation is done via GNU Make and a Python orchestration script. The recommended hardware is a x86_64 Linux machine with at least 16GB RAM, four CPU cores, Docker & GNU Make.

The artifact outputs all the metrics used in precision, performance, and scalability results reported in the paper. To validate the results, the benchmarks are run with different configurations and output metrics are compared against expected values.

A.2 Artifact check-list

- **Algorithm:** Building Compressed Symbolic Execution Graph, MHP analysis for Java
- **Program:** Java analysis written in Soot framework
- **Compilation:** Maven build using Eclipse Temurin 8 JDK inside Docker
- **Binary:** self-contained JARs containing all dependencies
- **Data set:** DaCapo materialized, Renaissance fj-kmeans, custom microbenchmarks
- **Run-time environment:** Eclipse Temurin 8 JRE within Docker
- **Hardware:** >= 16 GB RAM, 4+ cores recommended
- **Metrics:** per-benchmark CSVs recording precision, runtime, memory, graph size, number of joins
- **Output:** `docker-logs/` directory containing logs and metrics
- **Experiments:** reproduce precision, scalability, and performance tables and compare multiple MHP methods
- **Disk space:** 5GB (image + datasets + logs)
- **Time to prepare workflow:** 15 minutes (pull image, `make build`)
- **Time to complete experiments:** Most experiments complete within 40 minutes
- **Publicly available:** Yes
- **Workflow automation framework used:** GNU Make + Python orchestration inside Docker

A.2.1 How to access. Download the artifact tarball from 10.6084/m9.figshare.30928775. The artifact tarball contains the code, benchmarks, and Docker environments to build and run the artifact. All commands are executed from the root of the extracted directory.

A.2.2 Hardware dependencies. It is recommended to use an x86 machine with at least 16 GB RAM and four or more physical cores

A.2.3 Software dependencies. Docker and GNU Make are required on the host system. Other dependencies are captured within the Docker image built from the provided Dockerfile.

A.2.4 Data sets. All datasets reside under `Data/`. DaCapo & Renaissance benchmarks are boosted/materialized with TamiFlex to handle reflection. The benchmarks are located under `Data/DaCapo/` and `Data/Renaissance/` respectively. Custom microbenchmarks live under `Data/SelfWritten/`.

A.3 Installation

Extract the artifact tarball to a local directory. Run `make build` from the root of the extracted directory. This step installs the exact JVM toolchain and builds the analysis binaries inside the container image.

A.4 Experiment workflow

Run `make shell`. This mounts `Data/`, `docker-logs/` and drops you into a shell prompt inside the Docker container.

Run the orchestration script `./run_benchmarks.py` inside the shell. `./run_benchmarks.py -help` displays all available options.

Some example invocations are listed below:

- List available methods and benchmarks:
  ```
  ./run_benchmarks.py --list
  ```
- Run a built-in DaCapo benchmark:
  ```
  ./run_benchmarks.py --mhp_method GRIP_MHP
  --benchmark h2 --inlining-mode full
  ```
- Execute the LV baseline on `sunflow` with targeted inlining:
  ```
  ./run_benchmarks.py --mhp_method LV
  --benchmark sunflow --inlining-mode targeted
  ```
- Launch a custom benchmark rooted under `Data/`:
  ```
  ./run_benchmarks.py --mhp_method GRIP_MHP
  --data-path Data/CustomTest/ --entry-class
  CustomTest.Main --inlining-mode full
  ```

To accommodate longer runs if needed edit the timeout variable `TIMEOUT_MINUTES` in the Makefile before spawning the shell (Default is 60 minutes).

A.5 Evaluation and expected results

Each invocation emits logs and metrics CSV files under `docker-logs` on the host. The logs are organized by MHP method and then log type (RunLog or Metrics). The name of the files is the benchmark name appended with the timestamp.

Running the full suite reproduces the original data within expected measurement noise.