A Practical MHP Information Analysis for Concurrent Java Programs

Lin Li and Clark Verbrugge

School of Computer Science, McGill University
Montréal, Canada
{lli31,clump}@sable.mcgill.ca

Abstract. In this paper we present an implementation of *May Happen in Parallel* analysis for Java that attempts to address some of the practical implementation concerns of the original work. We describe a design that incorporates techniques for aiding a feasible implementation and expanding the range of acceptable inputs. We provide experimental results showing the utility and impact of our approach and optimizations using a variety of concurrent benchmarks.

1 Introduction and Motivation

Although specific techniques for handling problems related to compiling multithreaded languages are being actively researched, e.g., synchronization removal [7], and race detection [4], more general techniques that also allow one to compute the impact of concurrency on other compiler analyses or optimizations are still desireable. Such a more general approach for Java is provided by Naumovich et al's *May Happen in Parallel* (MHP) analysis [15]. This analysis only determines which statements may be executed concurrently, but from this information on potential data races and synchronization problems can be derived.

The original MHP algorithm relies on a simplified program structure. All methods need to be inlined, and cloning is necessary to eliminate polymorphism and aliasing. Unfortunately, while these limitations still allow a variety of applications to be analyzed, they cannot be feasibly applied to more complex programs. Whole program inlining is not possible for non-trivial programs, and moreover excludes many recursive programs. Cloning further expands the program size, and even in the presence of good alias resolution is likely to cause space concerns. Thus although Naumovich et al's results are encouraging, it is important to also know how well the analysis would work in a more practical compiler setting.

In this paper we present an implementation of MHP for Java that attempts to address such practical concerns. Our implementation of MHP incorporates several simple analyses as well as modifications to MHP structures in order to reduce many of the practical limitations. We provide experimental results and show how simple optimizations on the MHP internal data structures can make MHP analysis of even moderate size programs quite feasible.

In the next section we describe the basics of Naumovich et al's MHP analysis and its accompanying *Parallel Execution Graph* data structure. Further details on our implementation and context are given in section 3. Improvements to this implementation are then developed in section 4, and experimental results and analysis are given in section 5. Related work is described in section 6 and we describe future work and conclude in section 7.

2 PEG and MHP Analysis

MHP analysis first requires the construction of a *Parallel Execution Graph* (PEG) data structure, an augmented control flow graph for the input program. The actual analysis is then on the PEG, with a fairly trivial mapping back to the original CFG. We sketch out the major steps and structure definitions here; complete details are of course provided in Naumovich et al's original paper [15]. First, however, we give further details on the practical constraints.

MHP analysis relies on a simplified and constrained input program structure, including limits on thread creation, method and variable aliases and method call structure. Some constraints such as having a known and bounded number of runtime threads represent reduced generality, but have no impact on efficiency. Others however imply significant cost, and severely impact practicality.

One main requirement is that alias resolution be done, and code cloning used to eliminate polymorphism and ensure precise variable and method targets are known. This simplifies the analysis at a potentially very large cost in data size and thus overall running time. More complex programs with larger alias sets cannot be efficiently represented or analyzed under these constraints.

MHP analysis is not defined over most method calls, and requires all methods except specific *communication methods* (`Thread.start(), wait(), notify()` etc) to be inlined. This eliminates the need to consider issues of disentangling information propagated back from multiple call sites to the same callee (the *calling context problem*). However, recursive programs cannot then be analyzed without prior conversion to iterative forms. More critically, and particularly in conjunction with cloning, the space requirements of this approach can easily be excessive for even moderate programs, and so is not feasible in general.

2.1 Parallel Execution Graph

The *Parallel Execution Graph* or PEG is a superstructure of a normal control flow graph. Special arrows and nodes are incorporated to explicitly represent potential thread communication and synchronization. Since thread bounds are known, the actions of each thread are also uniquely represented in the graph. Figure 1 gives an example of a PEG for a simple program that launches 2 threads (`t1` and `t2`) from a main thread and then attempts to signal them using a global lock and a wait/notify pattern.

Nodes in PEG's are structured as triples; e.g., for *communication method*s the triple *(object, name, caller)* is used, where the field *object* represents the monitor

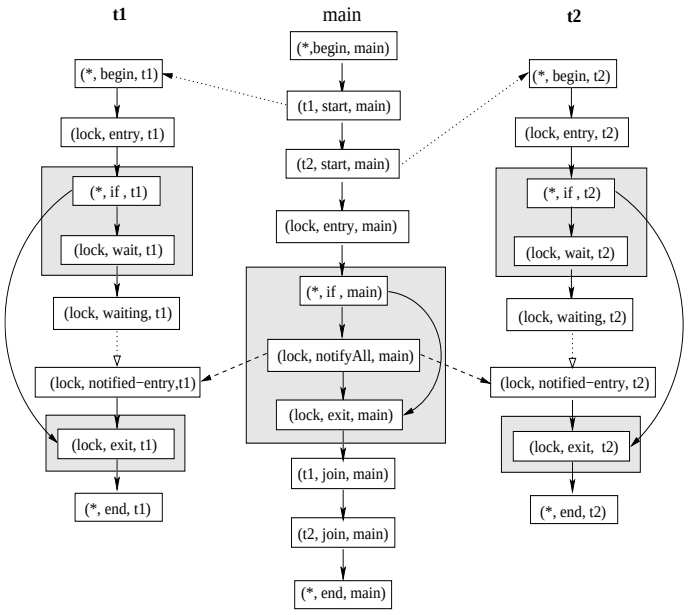

Fig. 1. An example of a PEG, a simplified version of figure 3 in [15].

object controlling the communication, *name* is the method name, and *caller* is the thread name. For nodes that do not represent *communication method*s, a wildcard symbol (*) is used for the object field.

Certain new nodes are added to aid in later analysis. Most simply, *(*,begin,t)* and *(*,end,t)* nodes are inserted to mark the beginning and end of each thread *t,* and *(lock, entry,t)* and *(lock, exit,t)* nodes indicate monitorenter and monitorexit operations for operations by *t* on object *lock.* Condition synchronization is only slightly more complex. A `wait()` method call is broken down into a chain of *wait*, *waiting* and *notified-entry* nodes, representing the substeps of starting the call to `wait()`, actually sleeping after the lock is released, and having been notified and trying to reacquire the lock, respectively.

PEG edges fall into one of four different categories: *local*, *start*, *wait* and *notify* edges. The first three are statically constructed, and the last is created during the analysis. A *local edge* represents normal, intra-thread control flow, not dependent on thread communication. These edges are inherited from the base CFG, and are shown as solid edges in Figure 1. A *start edge* is created to indicate a must-precede relation between a call to `Thread.start()` and the first action of the initiated thread. These edges are shown in Figure 1 as the dotted edges with solid arrowheads between the *(t_i,start,main)* nodes and the corresponding *(*,begin,t_i)* node. A *waiting edge* models the control flow dependent on thread notification. These are inserted between *waiting* nodes and *notified-entry* nodes, and are shown as dotted edges with empty arrowheads in Figure 1.

Notify edges are created dynamically during the analysis process. They allow precedence information to flow from the notifier to the waiting thread, and since they are inserted during analysis, this information flow can be more precise than a static approach. Notify edges are only inserted from an *(object,notify/notifyAll,t1)* node to a *(object,notified-entry,t2)* node if the same object is involved, the threads are distinct, and the analysis has computed that these two events may indeed happen in parallel.

2.2 A Worklist Flow Analysis Algorithm

MHP analysis is performed using a worklist dataflow algorithm. The goal is to find for each PEG node the set of other PEG nodes which may execute concurrently. For each PEG node a set $M(n)$ is initialized to the empty set, and a least fixed-point based flow algorithm propagates set information around the PEG. Although this largely follows the template of a standard dataflow analysis, with special modifications to create notify edges and flow information across and through the various special edges and nodes, the algorithm also includes a "symmetry step" to guarantee that if $m \in M(n)$ then $n \in M(m)$. This non-standard component of the analysis ensures information is accurately maintained as the actions of concurrently executing threads are analyzed. Note that as with most static analyses the computed information is a conservative approximation.

3 MHP Analysis in the context of Soot

3.1 Soot Framework

Our implementation is based on Soot [22], a free compiler infrastructure written in Java. The Soot framework was designed to provide a common infrastructure for analyzing and transforming Java bytecode, and in particular includes a number of useful analyses, transformations and representations we used to simplify our effort. Major components are described below.

Jimple The main internal program representation in Soot is *Jimple.* Jimple is a typed, "3-address" code representation of input, stack machine based bytecode, and Soot provides control flow graph construction and various control flow analyses on Jimple. Since a stack-less, CFG form is also convenient for MHP analysis, we based our analysis on Jimple. This also simplifies interaction with other analyses in the Soot framework.

Intra-procedural Analysis Soot has two built-in intra-procedural analysis schemata: `ForwardFlowAnalysis` and `BackwardFlowAnalysis`. Due to the symmetry step MHP analysis is strictly speaking neither a forward flow analysis nor a backward flow analysis; we implemented our MHP analysis based on the ForwardFlowAnalysis framework, modified to incorporate the symmetry step.

Inter-procedural Analysis Soot also provides several inter-procedural analyses important to our implementation:

- **Call Graphs** For a multithreaded program, the *CallGraph* must include all the methods that can be reached from the `main` method, as well as the `run` method in a class that implements `java.lang.Runnable.`
- **Class hierarchy analysis (CHA)** *Class hierarchy analysis* [5] conservatively estimates the run-time targets of method calls by using the class-subclass relationships in the type hierarchy.
- **Points-to analysis** *Points-to* analysis [6] computes the set of concrete locations to which each variable may point. Points-to information identifies variable aliasing, and in object-oriented languages like Java, method targets too. Soot includes SPARK [14], a points-to analyzer that provides fast and precise points-to data.

Figure 2 shows how our MHP analysis is integrated with Soot. Java class files are first input into Soot framework, producing Jimple files, the Call Graph, as well as CHA and Spark analysis information. These are all used as input to the MHP module, which computes the may happen in parallel information for each PEG node. MHP information can subsequently be used for further program analyses and optimization.

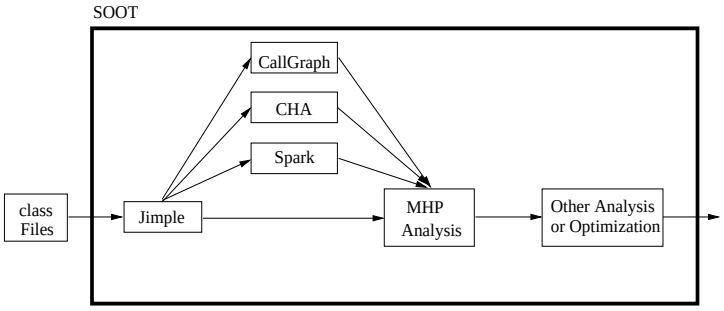

Fig. 2. MHP Analysis in Soot.

4 Practical MHP Analysis

Our MHP implementation is composed of a few steps; Figure 3 shows an overview of the process. There are three phases in our MHP analysis. The first phase is a PEG Builder which uses Jimple and takes input from CallGraphs, CHA, and SPARK. We get PEGs after the PEG builder phase, then a PEG Simplifier works on PEGs to get a smaller PEG by aggregating some nodes into one node. The final phase is an MHP analyzer which runs the worklist algorithm based on

the simplified PEG. Each of these processes has performance-affecting practical considerations or goals, and we describe the salient features below.

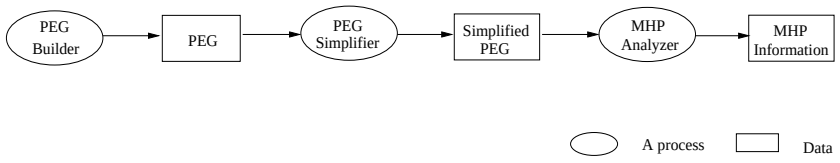

Fig. 3. Overview of our MHP analysis.

Note that many of our simplifications are based on the observation (made in [15]) that code not containing synchronization does not need to be explicitly modelled. Here, by *interesting statements* we refer to statements related to modeling execution of threads and synchronization of Java programs, i.e., the communication methods `wait()`, `notify()`, `notifyAll`, `Thread.start()`, and `Thread.join()`, as well as *monitorenter*, *monitorexit* bytecode operations (including entry/exit of synchronized methods). A method is interesting if it either contains an interesting statement, or any callee is interesting.

4.1 Efficient PEG Construction

Conceptually, building PEG's from a CFG is straightforward. In practice, non-obvious information needs to be computed to make correct decisions. In order to keep the data size manageable, a realistic implementation must also incorporate techniques to limit the size of the resulting data structures.

One obvious way of restricting data size is to focus attention on application code only. Java includes a very large standard class library, and so even for a very small program a complete call graph tends to be quite large. However, in many cases the application itself is of main interest, and so if external actions are assumed safe enough, greater efficiency can be derived by excluding library and startup information. We therefore define a *PegCallGraph* to be a call graph restricted to methods inside application classes, i.e., user defined classes.

Constructing the PEG can involve a lot of duplicated effort, as the same method is inlined in various places. Our strategy is to build small PEGS, one for each method a thread may invoke, and then combine these small PEGs into a PEG for the whole program. This of course doesn't change the final PEG size, and other techniques are necessary for that. Methods without *interesting statements* are good candidates for pruning, and so our PEG construction first proceeds with a simple, fast interprocedural analysis to identify and compact such methods, followed by a standard inlining operation.

Finding Interesting Methods Clearly methods that will never execute any interesting statements are of little interest to the MHP analysis: any MHP in-

formation true on entry to such a method is true at exit, and at all points in between. Since thread communication code is typically a small part of any significant program, restricting the PEG to useful parts of the program is very effective.

Unfortunately, knowing whether a method is interesting is recursively dependent on the status of all callee methods. A precise, flow-sensitive interprocedural analysis would be most effective, but is of course both complex and expensive. We have elected for a more pragmatic flow-insensitive approach, implemented in two stages.

The body of each method in the PegCallGraph is first scanned to see if contains an interesting statement. If so the method node in the PegCallGraph is marked interesting. Once all methods are examined, marks are propagated in the reverse direction of call graph edges, and logically OR'd at each merge point using a depth first search of the PegCallGraph. The result is a conservative overapproximation of interesting methods. During actual PEG construction uninteresting methods are represented by single node placeholders, greater reducing PEG size.

Recursive method calls will result in the failure of inlining, and so naturally must be avoided. The call graph is thus also analyzed to locate recursive cycles, and rejects the input program if so. Of course cycles are only problematic if interesting methods are involved, since uninteresting methods are not actually included in the PEG. Our algorithm ensures any detected call graph cycles involve at least one interesting method before rejecting the program.

Inlining Actual inlining is straightforward, and proceeds in a bottom up fashion on the PegCallGraph. Each inlining operation involves creating a new local scope for the code and mapping local variable, parameter and return value usage. In the case of Java, care must also be taken to ensure appropriate *monitorenter* and *monitorexit* instructions are inserted in the case of inlining `synchronized` method calls.

Note that because inlining is used in computing MHP information, finding precise method targets is very important. Imprecision in the destination of virtual calls can have a large impact on call graph size. To get a more precise call graph than that provided by CHA alone, we used Spark to help resolve objects used in invocation calls, and hence method polymorphism. In places where the method target was still ambiguous all potential callees must be presumed invoked.

4.2 PEG Simplification

We can proceed to use the MHP algorithms to compute MHP information once the PEG is built. However, even with the above inlining strategy we may still have a large PEG. Further optimization techniques can still be useful to simplify the PEG before running the MHP algorithms, and so we applied two straightforward graph reductions as optimizations: merging lists, and collapsing strongly

connected components. Since the MHP analysis manipulates sets of PEG nodes, reductions in PEG size can have a significant effect, and we give some results on the effect of PEG reductions in Section 5.

Merging Strongly Connected Components This is based on an observation: suppose a *strongly connected component* (SCC) S inside a PEG does not contain *interesting statements*. If a statement A can be concurrently executed with a statement B inside S, it should also be possible for A to be concurrently executed with all the other nodes inside S. Thus, we can merge the nodes inside this SCC and create a new node to represent the entire SCC. The new node is simply a reference to the list containing all the nodes inside the SCC.

After finding SCCs, we check if they contains *interesting statements*. If not, we can merge the nodes in the SCC into one node.

Merging Sequential Nodes A sequence of nodes with no *interesting statements*, and no branching in or out except at the beginning and end respectively necessarily has the same MHP information at each node in the sequence. We thus locate all maximal chains of this form, and as with SCC's collapse them into a single node. Again, these new nodes are references to lists of the replaced nodes.

4.3 Practical MHP Analysis

The efficient PEG construction described above incorporates inlining, but avoids resolving variable aliases through cloning. The latter technique is quite expensive in an allocation-intensive setting such as Java.

Specifically handling object aliases in the MHP analysis would significantly complicate the algorithm, and certainly increase its actual running time. It is further unclear whether this extra effort is worthwhile, given that even a set of 2 potential object targets for a monitor operation may make a conclusion of success or failure of the operation impossible. We have thus chosen to focus on detecting situations in which precise conclusions can be made rather than on a general inclusion of aliasing. Below we describe our technique for handling this problem..

Finding Runtime Target Objects MHP analysis relies on knowing the value of the *Object* field in PEG triples for determining lock ownership and monitor-based information flow. In Soot and by using SPARK, it is possible to find the potential textual allocation sites corresponding to a given object reference. Allocation sites are locations in the code, and thus one can easily determine a set of potential types of an object reference, and this is sufficient for many analyses (including call graph refinement).

For MHP analysis, however, decisions as to whether synchronization has occurred requires knowing that an object involved in a *monitorexit* is the same *runtime* object involved in a previously examined *monitorenter*. SPARK computes

may-alias information, and so even the same singleton allocation site sets for the respective objects are not sufficient for this conclusion, since allocation sites in loops may spawn more than one runtime object. A form of interprocedural value numbering analysis is thus required. Again for simplicity of implementation and as well as asymptotic complexity concerns we have elected for a custom analysis, composed of an intraprocedural analysis and a flow-insensitive interprocedural step.

An allocation site that is only ever executed at most once of course does represent one runtime object. Thus an obvious guarantee that two or more synchronization operations are operating on the same value can be provided if the computed sets of allocation sites are both the same singletons, and the allocation site is only ever executed once.

Intraprocedurally, a statement is surely executed at most once if it is not included in any control flow cycles, and so is the complement of knowing what may be executed more than once. This information is computed for each allocation site of every method in the PegCallGraph. To find out which methods are called more than once interprocedurally, we use a modified depth-first search on the PegCallGraph to detect whether a node is potentially reachable more than once from `main`. Methods that can be called more than once conservatively imply each statement in them can be executed more than once, regardless of internal control flow. Our algorithm actually computes both intra and interprocedural information together, performing intraprocedural analysis as the interprocedural analysis proceeds, and only if required. This allows the conclusions of each analysis to be merged and propagated together.

Finding Monitor Nodes Computation of MHP information is partially based on knowing which PEG nodes may be contained within a Java monitor lock. However, as well as the need to determine the exact runtime identity of a locked object, an analysis of Java locks must also account for recursive locking—a thread that owns a lock may relock it repeatedly, and is required to unlock it a corresponding number of times in order to release it. Simply identifying nodes dominated by an enter node without reaching an exit node is thus insufficient to determine whether a node outside this region is or is not protected by a monitor—lock level must also be tracked.

To model locking state we have implemented a simple, forward, flow-sensitive analysis on the PEG. This analysis conservatively tracks locking depth for objects used in monitor operations by associating a lock count with each such object. These structures are propagated through the PEG, incrementing the count for the object specified at each *monitorenter* operation and decrementing counts at *monitorexit*'s. Unbounded recursive locking, as in general merge points with unmatched locking depths for corresponding objects (not possible with Java programs) and are not handled, so this is guaranteed to reach a fixed point.

With lock depth information the MHP analysis can make sound judgements as to whether a PEG node is truly in a monitor or not.

5 Experimental Results

5.1 Benchmarks

We collected our benchmarks from several sources. Most of the benchmarks are multithreaded benchmarks from the Java Grande Benchmark Suite [21]: ForkJoin, Sync and Barrier represent low level benchmarks that test synchronization, Series, LuFact, Sor, Crypt and SparseMult test specific "kernel" operations, and MonteCarlo, RayTracer and MolDyn are larger, more complete applications. mtrt is the only multithreaded benchmark from the SPECjvm98 [1] suite. In order to fit our input requirements, we modified most of these benchmarks by manually unrolling all the loops containing method calls to *communication method*s.

For comparative purposes we have also attempted to collect some of the same benchmarks used in Naumovich et al's paper. However, most of the code we have been able to acquire is in the form of incomplete program fragments that require a driving main program to analyze in our system. Fine-grained comparisions are thus not likely to be meaningful. We therefore include AuBanking and PeBanking, programs based on the examples AutomatedBanking and PessimBankAccount from Doug Lea's book [11]. We have focussed on these two examples since in [15] Naumovich et al's version of these benchmarks had the largest PEG sizes and also had the largest MHP analysis times (by an order of magnitude) of all their benchmarks. Cyclic is a smaller benchmark from the CyclicBarrier example in the second edition of Lea's book [12]. In each case we added an appropriate main method, modifying them to be complete applications. All tests were run on a Pentium 4 1.8GHz, using the Sun HotSpot VM 1.4.1 (maximum 1500Meg heap) under Debian Linux.

5.2 Results

Tables 1 and 2 present the experimental results of our MHP analysis. In Table 1 the first column gives the names of the benchmarks, the second column gives the number of threads (including the main thread), and the next two columns give the number of nodes and edges in the PEGs representing each program respectively. In the fifth and sixth columns, we specify the average and maximal number of nodes in the computed $M()$ set for each node, i.e., how many nodes were determined may be executed in parallel with each node. This gives some notion of analysis accuracy, at least in the absence of measuring a consuming analysis. The seventh column gives the total number of node pairs found in the entire PEG—as well as the PEG itself, this represents the total space requirements of the analysis.

The remaining columns measure time for the various stages of the analysis. PEG time is the time to build the PEG, MHP is the subsequent analysis time, and Spark time is the total cost of points-to analysis. Total time is greater than the sum of the these stages; the remainder represents time required to load and initialize and shutdown the Soot environment.

The timings and data in Table 1 already represent application of many of the previously discussed simplification and implementation techniques (excessive data sizes prevented computation of totally unoptimized data), we only exclude the PEG node merging techniques of Section 4.2. Note that MTRT contains recursive method calls. Method inlining for such a benchmark would normally fail; however, using the techniques of Section 4.1 we determined that the recursive calls do not involve *interesting statements*, and so we are still able to get results.

For most benchmarks the time to build the PEG is small, and in all but one case well under a second. MHP analysis time clearly dominates PEG construction time. This is unsurprising given the $O(n^3)$ time complexity of MHP analysis, but was considerably less evident in the data presented in [15], where the majority of benchmarks were very small (mostly < 100 PEG nodes) and so PEG time generally appeared to dominate. For larger programs the cubic behaviour of MHP becomes more evident: MOLDYN, the largest benchmark we examined at 2173 nodes takes less than 2 seconds to build the PEG, but over 12 hours to analyze. These running times are clearly still excessive for even moderate programs, and further steps are necessary to reduce PEG size, and thus MHP analysis time.

Programs	Threads	Nodes	Edges	$\|M()\|$		Pairs	PEG	MHP	Spark	Total
				Ave	Max		(s)	(s)	(s)	(s)
FORKJOIN	4	308	331	64	173	6105	0.18	4.46	67.2	88.5
SYNC	5	656	712	118	459	28944	0.40	51.51	68.2	136.8
BARRIER	5	561	716	175	339	34651	0.34	72.72	68.7	160.4
CRYPT	5	1025	1061	672	772	297220	0.52	6812.68	67.2	6917.7
MONTECARLO	3	405	433	104	182	11340	0.28	14.15	68.0	102.3
RAYTRACER	3	660	724	125	318	25188	0.37	57.58	67.5	143.42
SERIES	3	315	342	109	130	9660	0.24	8.84	67.8	93.3
LUFACT	3	465	510	202	224	32032	0.23	87.86	68.8	163.08
SOR	3	662	673	289	363	66430	0.29	259.26	68.0	347.9
SPARSEMULT	3	305	329	81	120	6180	0.21	3.98	67.2	88.1
MOLDYN	3	2173	2295	1093	1866	1088392	1.86	44313.44	69.2	44553.9
CYCLIC	5	162	201	69	124	4580	0.14	1.13	67.8	86.2
MTRT	4	188	211	43	108	2819	0.33	1.53	139.7	232.9
AUBANKING	3	170	203	31	92	4114	0.17	1.14	66.5	86.4
PEBANKING	3	154	270	63	137	4414	0.14	1.17	66.4	85.3

Table 1. Experimental results without PEG simplification

Table 2 shows similar experimental results when the PEG is optimized using the techniques of Section 4.2. The second and third columns give the PEG size reductions supplied by the two techniques of merging SCCs and merging sequential nodes respectively; the resulting graph size is given in the fourth and fifth columns. In smaller programs sequential node contractions are most effective,

Programs	Sim.Scc	Sim.Seq.	Nodes	Edges	Sim. (s)	MHP (s)	Total (s)	Total Speedup	PEG+MHP Speedup
ForkJoin	0	199	109	132	0.02	0.41	84.4	1.05	4.76
Sync	2	389	255	307	0.07	8.81	94.1	1.45	5.95
Barrier	12	287	262	411	0.06	21.21	108.8	1.47	3.71
Crypt	662	240	121	149	0.10	0.93	105.1	65.82	4395.80
MonteCarlo	26	247	132	158	0.03	0.53	88.7	1.15	17.13
RayTracer	18	431	211	267	0.07	6.66	92.5	1.55	8.48
Series	26	180	109	134	0.03	0.64	85.0	1.09	9.98
LuFact	166	194	105	130	0.04	0.53	87.9	1.91	110.06
Sor	298	223	101	124	0.04	0.39	89.0	3.91	360.37
SparseMult	55	165	85	104	0.02	0.09	84.5	1.04	12.65
MolDyn	1482	547	144	174	0.18	1.18	90.0	495.04	13763.80
Cyclic	0	51	11	150	0.02	0.74	85.8	1.00	1.40
mtrt	3	107	78	95	0.02	0.10	231.8	1.00	3.73
AuBanking	2	71	97	126	0.02	0.53	85.8	1.01	1.75
PeBanking	0	66	88	204	0.02	0.62	84.7	1.01	1.68

Table 2. Experimental results after optimization

but in the bigger programs the volume of modular, synchronization independent sections of code sometimes made SCC merging quite valuable. In every case our PEG optimizations were able to reduce the graph, and in some cases quite dramatically: MolDyn is reduced from 2173 nodes to 144.

The next two columns give the time in seconds taken to perform the PEG simplifications and run MHP analysis on the smaller PEG. The eighth column shows the total running time including Spark and Soot overhead. The remaining columns give the relative speedup (old-time/new-time) ratio achieved by the optimized version versus the base approach, for both total running time, and the time just to construct and simplify the PEG and run the MHP analysis. Again, MolDyn speedups were most significant, as running time drops from half a day to just under 2 seconds. As a general rule, larger benchmarks have more nodes, and hence more opportunities for PEG compaction, which is quite encouraging for analysis of reasonable size programs. The benchmarks with the lowest speedup, Cyclic, AuBanking and PeBanking, also have the fewest reductions due to PEG simplification, both in absolute terms and proportionally. These are also all relatively small benchmarks with a high proportion of communication and synchronization statements, and this limits merging opportunities.

SCC and sequential merging has clear benefits, with a fairly minimal cost—even for MolDyn simplification takes less than 1/5s. Merging in combination with an already efficient initial PEG construction allows reasonable size programs to be analyzed. Interestingly, after optimization efforts, the Barrier benchmark is the most expensive to analyze, and MolDyn time is even less than Sync. With optimization overall analysis cost is related more closely to number and density of communication operations than input program size.

6 Related Work

Obviously, our work here is based most directly on the MHP analysis originally designed by Naumovich et al [15]. There are of course other approaches to analyzing and representing concurrent programs, with a variety of specific and general purposes.

Program Dependence Graphs (PDGs) [8] can be used for general program optimizations where dependency is a concern; for example, detecting medium to fine-grain parallelism for sequential programs. They are however not designed to represent parallel programs. *Parallel Program Graphs* (PPGs) [18, 19] are a generalization of PDGs and CFGs and can be used to fully represent sequential programs and parallel programs. PPGs can be used for program optimization and detecting data races. Srinvasan et al. [10] proposed a *Parallel Flow Graph* (PFG) for optimizing explicitly parallel programs. They provided dataflow equations for the reaching definitions analysis and used a copy-in/copy-out semantics for accessing shared variables in parallel constructs. *Concurrent Control Flow Graphs* (CCFGs) [13] are similar to PPGs and PFGs, with the addition of conflict edges in addition to synchronization and control flow edges. None of these representations are Java-specific.

Specific problems have engendered more specific, and more efficient results. For the purpose of data race detection, Savage et al developed *Eraser*, a race checker in the C, C++ environment. Jong-Deok Choi et al [4] compute relatively precise data race information using *Inter-Thread Control Flow Graphs* (ICFGs). Flanagan and Freund analyze large Java program for race conditions by examining user-provided type annotations for code [9]. Improvements to accuracy and efficiency of data race detection continue to be addressed; e.g., through dynamic techniques [23], and by combining information from multiple analyses [16]. A similar concentration of efforts has looked at synchronization removal [3, 7].

Our implementation and optimization techniques largely depend on a combination of well known approaches. Good quality points-to analysis is one of the more complex and expensive compiler problems, and has been addressed in a variety of settings [2, 6, 17, 20]. Spark [14] produces precise points-to information, and this has been quite crucial to our ability to analyze non-trivial programs. Exclusion and compaction of PEG nodes according to the presence of communication methods was briefly mentioned, though not developed in [15].

7 Future Work and Conclusions

We have presented a more realistic implementation of MHP analysis for Java. Our design makes use of a variety of existing and small custom analyses in order to build a feasible implementation that can analyze programs of a reasonable size, bypassing a number of previous input restrictions. We have presented experimental results from such an implementation, and shown how excessive MHP analysis time can be efficiently handled through simple input compaction techniques.

Our work has clear extensions in a number of ways, including analysis and potential implementation improvements. Certainly accuracy of the resulting information deserves examination. Naumovich et al compare MHP information to precise reachability analyses, but this is not feasible for larger programs. Accuracy could however be judged by assessing how useful the information is to a consumer analysis, such as race detection or synchronization removal.

Internal improvements can of course still be done. Our simple value prediction and interesting method identification algorithms are sufficient to produce results, but are not especially precise. More accurate strategies could be applied, which would allow determination of the relative cost versus benefit for this information. Similarly, further PEG compaction approaches seem worth exploring.

We also aim to expand the range of acceptable input programs. Programs with an unbounded number of threads, use of timed synchronization constructs, and so on could be handled, and this would allow more programs to be analyzed with less manual intervention.

Acknowledgements This work has been supported by the National Sciences and Engineering Research Council of Canada, and the McGill Faculty of Graduate Studies. We would like to thank Ondřej Lhoták for lots of implementation help and advice.

References

1. SPEC JVM98 Benchmarks. http://wwww.spec.org/jvm98.
2. Marc Berndl, Ondřej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 103–114. ACM Press, 2003.
3. J. Bogda and U.Holzle. Removing unnecessary synchronization in Java. In *Proceedings of the ACM SIGPLAN 1999 Conference on Object-Oriented Programming, Systems, Languages, and Application*, pages 35–46, November 1999.
4. Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan Vivek Sarkar, and Manu Sirdharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, Berlin, Germany, June 2002.
5. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecure Notes in Computer Science*, pages 77–101, Åarhus, Denmark, 7-11 August 1995. Springer.
6. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 242–256, 1994.
7. E.Ruf. Effective synchronization removal for Java. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming language design and implementation*, pages 208–218, June 2000.

8. Jeanne Ferrante, Karl J.Ottenstein, and Joe D. Warren. The program dependence graph and its uses in optimization. In *ACM Transactions on Programming Languages and Systems*, July 1987.
9. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232. ACM Press, 2000.
10. Ferrante J, K.Ottenstein, and J. Warren. Compile-time analysis and optimization of explicitly parallel programs. In *Journal of Parallel algorithms and applications*, 1997.
11. Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, 1997.
12. Doug Lea. *Concurrent Programming in Java Design Principles and Patterns*. Addison-Wesley, Reading, Massachusetts, second edition, 1999.
13. Jaejin Lee. *Compilation techniques for explicitly parallel programs*. PhD thesis, University of Illinois at Unbana-Champaign, 1999.
14. Ondřej Lhoták. Spark: A flexible points-to analysis framework for Java. Master's thesis, McGill University, December 2002.
15. Gleb Naumovich, George S.Avrumin, and Lori A.Clarke. An efficient algorithm for computing MHP information for concurrent Java program. In *Proceedings of the 7th European engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, Toulous, France, 1999.
16. Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 167–178. ACM Press, 2003.
17. Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 43–55. ACM Press, 2001.
18. Vivek Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph representation. In *Proceedings of the 10th International Workshop on Languages and Compilers for Parallel Computing, LNCS Springer-Verlag*, Minneapolis, MN, August 1997.
19. Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In *Proceedings of ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Montreal, Quebec, Canada, 1998.
20. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41. ACM Press, 1996.
21. Java Grande Benchmark Suite. http://www.epcc.ed.ac.uk/javagrande/javag.html.
22. Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a Java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.
23. Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 70–82. ACM Press, 2001.