Understanding Parallelism-Inhibiting Dependences in Sequential Java Programs

Atanas Rountev*, Kevin Van Valkenburgh†, Dacong Yan* and P. Sadayappan*
*Ohio State University
†Eaton Corporation

***Abstract*—Many existing sequential components, libraries, and applications will need to be re-engineered for parallelism. This work proposes a dynamic analysis of sequential Java programs that helps a programmer to understand bottlenecks for parallelism. The analysis measures the parallelism available in the program by considering a hypothetical parallel execution in which the code within a method executes sequentially, but each caller will execute in parallel with its callees. A best-case scenario is assumed: every statement executes as early as possible, as long as all dependences from the sequential program are satisfied. The idealized speedup under this model is a measure of the method-level parallelism inherent in the program, independent of hardware and JVMs. The analysis employs bytecode instrumentation and an online algorithm which tracks the reads and writes of relevant memory locations during a run of the sequential program.**

If the best-case parallelism is low, this likely means that the program cannot be easily re-engineered into a scalable parallel version. In experiments with 26 Java programs, we observed this situation for most programs. This problem is sometines due to programmer decisions that were perfectly reasonable for a sequential program but would be detrimental to the performance of any parallel version. To pinpoint such decisions, we propose an approach that employs the dynamic analysis to automatically find memory locations whose read and write operations decrease the available parallelism. In three case studies, we demonstrate how these bottlenecks can be identified and eliminated using the proposed approach.

I. Introduction

With the emergence of multi-core processors as the computing engine in all commodity platforms—laptops, desktops, and servers—the computing community faces a great software evolution challenge. Existing applications must be parallelized if they are to experience any performance benefits from newer generations of processors. While not all legacy software will need to evolve, it is inevitable that many existing components, libraries, and applications will need to be re-engineered for parallelism. Relying only on fully-automatic techniques for parallelization is wishful thinking—ultimately, human insights will be needed for effective software evolution. Software tools can dramatically reduce the cost of this human involvement. Increased programmer productivity through powerful software tools is especially desirable due to the magnitude of the software parallelization problem and the wide diversity of programmers—unlike in the past, when the development of parallel programs was primarily a task for a small cadre of experts, it is now a problem that affects average programmers.

This work proposes a dynamic analysis of sequential Java programs that helps a programmer to understand bottlenecks for parallelism. This analysis is intended to be used as a first step in a parallelization effort, providing insights that quickly pinpoint program features and idioms that must be modified if there is to be any hope for creating a high-performance parallel version of the program. The result of this effort is a semantically-equivalent sequential program for which, later, a good parallel version could be easier to obtain. The process of creating the actual parallel version is not the target of this work; complementary approaches (e.g., [6], [7], [30]) can assist with this effort.

Overview and motivation. The goal of the proposed dynamic analysis is to measure the potential parallelism in a given sequential Java program and to identify program features that decrease this parallelism. At a high level, the question being asked is the following: *In the best-case scenario, what speedup is possible with method-level parallelism?* The method-level parallelism model assumes that the code within a Java method executes sequentially, but any two methods will execute in parallel as long as all dependences from the original sequential program are satisfied. This is the same model as the *function-level parallelism* approaches from previous work (e.g., [3], [4], [8], [10], [18], [19], [29]). Although the approach can be easily adapted to model finer-grain parallelism (e.g., parallel execution of the iterations of a loop), our focus is on method-level parallelism, which is a natural choice for object-oriented software.

"Best-case scenario" refers to an optimistic model of parallel execution in which every statement s executes as early as possible: as soon as all statements on which s is dependent have completed, s is executed. During a run of the sequential program, the dynamic analysis uses bytecode instrumentation to compute this best-case parallel time for every executed statement. If N is the number of executed statements and T is the best-case parallel time for the program[1], the ratio N/T characterizes the ideal speedup that could be achieved by a parallel execution. Others have used the term *available parallelism* to refer to a similar concept for instruction-level parallelism [1]; we will use this term in the rest of the paper.

The optimistic model of parallel execution focuses on the inherent method-level parallelism in the given sequential Java program and filters out the effects of other factors such as limited number of processors, cost of creating and destroying threads, thread synchronization, thread scheduling policies,

[1] T is abstract time, with unit cost for each executed statement.

etc. The key goal is to consider the parallelism inherent in the computation, while suppressing all other factors.

Program understanding. The first intended use of the analysis is to understand aspects of the behavior of the sequential program that are relevant for future code parallelization.

Characterization of parallelizability. The available parallelism indicates how difficult it would be to create a high-performance parallel version of the sequential program. In particular, if the available parallelism is low, this likely means that the program cannot be easily re-engineered into a scalable parallel version. In this case, even if the programmer created the appropriate threads and synchronizations, even if *every* method call was executed in parallel with the caller method, the bottom-line result would still be negligible performance improvements. In experiments with 26 Java programs, we often observed this situation. Our studies of several program with this behavior revealed fundamental roadblocks to effective parallelization—sometimes due to decisions that were perfectly reasonable for a sequential program but would be detrimental to the performance of any parallel version.

This characterization also provides insights about the potential benefits of *speculative parallelization*. Several projects have considered speculative method-level multithreading support for Java (e.g., [3], [4], [10], [19]). In this context, the original sequential program is not explicitly parallelized. Instead, the speculation machinery optimistically executes a callee in parallel with the caller, and applies the appropriate correctness-preserving mechanisms when speculation fails. The available parallelism measured by the dynamic analysis is an indicator of the potential improvements due to such speculation.

Search for parallelism bottlenecks. Our experiments indicate that data dependences through fields of heap objects are the most significant sources of parallelism bottlenecks in Java programs. In a large program with hundreds or thousands of fields, identifying a small subset of problematic ones is a daunting task. Using an extension of the analysis, we propose an approach that automatically finds such bottleneck fields. In one variation of the approach, we consider the effects of ignoring all dependences through a particular field. The *increase* in available parallelism is used for ranking: the higher the increase, the more "suspicious" the field. This approach works well if there is a single problematic field. In the presence of multiple bottlenecks, a second variation of the approach is used. Here we consider the effects of respecting all dependences through a field, while ignoring all dependences through all other fields. Ranking is based on the *decrease* in available parallelism, compared to ignoring all dependences through all fields. As an example, in one of the case studies this approach identified five fields (from a total of more than fifty fields), each one of which was an independent bottleneck.

Program refactoring. The other intended use of the analysis is to judge the potential benefits of code transformations. As discussed earlier, we consider a two-stage parallelization effort; others have also taken a similar view [6]. First, the given sequential program is refactored into an equivalent sequential program with higher available parallelism; in the second stage, an actual parallel version is created. In the first stage, the programmer can experiment with various design decisions and code transformations, without having to create a fully-functioning parallel program. Various alternative designs can be explored with ease, and the promise of each design can be evaluated quickly by considering how much it increases the available parallelism. This exploration will be with respect to the intrinsic properties of the computation—essentially, the shape of the run-time dependences—and will not be obscured by effects from the hardware or the JVM.

This approach also provides valuable guidance in the presence of mechanisms for speculative parallelization. Although in this case the programmer does not need to create an explicitly-parallel version, she still must eliminate parallelism-inhibiting dependences—otherwise, they will have detrimental effect on the performance of speculative parallelization.

Contributions. The evolution challenges due to multi-core hardware require a very wide range of approaches. Some software components will need complete re-implementation, while others will evolve with more lightweight changes through sequences of refactorings [6]. For the latter scenario, our techniques can provide valuable assistance through a dynamic analysis that measures inherent parallelism (Section II) together with client analyses to characterize the effects of different memory locations and to search for problematic fields (Section III). We present an experimental evaluation on 26 programs (Section IV) as well as three case studies demonstrating how the analyses can help to uncover parallelism bottlenecks and to evaluate the effects of their removal (Section V).

II. Dynamic Analysis of Available Parallelism in Sequential Java Programs

Our approach is inspired by Kumar's COMET tool [12] (short for "*co*ncurrency *me*asurement *t*ool"). This work from the late 1980s considers a best-case parallel execution of a given sequential Fortran program. In such a best-case scenario, a statement s is executed as soon as possible, while respecting all dependences of s on statements that occurred before s in the run of the sequential program. Kumar's technique performs code instrumentation to introduce a *shadow location* x' for each memory location x, as well as new code statements to manipulate x'. During a subsequent run of the instrumented sequential program, x' is used to record the times of write and read operations on x for a hypothetical best-case parallel execution. At the end of the run, this recorded information provides a hardware-independent and compiler-independent upper limit on the statement-level parallelism available in the analyzed program. The available parallelism is quantified by N/T, where N is the number of executed statements and T is the best-case parallel execution time. The focus of this work are scientific and engineering applications written in Fortran. The model of parallelism is fine-grained (i.e., individual statements execute concurrently) and is suited for instruction-level parallelism (ILP), since the goal is to evaluate and improve the ILP capabilities of hardware and compilers. The approach is designed for analysis of arrays and scalars and

does not handle dynamically allocated memory, pointer-based data structures, and user-defined types.

Our goal is to define a dynamic analysis of available parallelism in Java programs. We are not interested in ILP considerations: the units of parallel execution are individual Java methods. Unlike Kumar's analysis and related approaches (e.g., [1], [9], [13], [14], [16], [20]–[22], [28]), we focus on finding code elements that need to be examined by a programmer for potential parallelism bottlenecks. By searching for suspicious entities, we can provide the user with insights that increase the productivity of her parallelization efforts.

Overview and example. To illustrate the analysis, consider the sequential program shown in Figure 1. This example is based on one of the case studies discussed later in the paper. We assume that the bytecode is transformed to some IR suitable for instrumentation; in our implementation, this is the three-address Jimple IR in the Soot framework [25]. The example uses simple Jimple-like statements (e.g., a constructor call is represented as a separate statement). The middle part of the figure shows the trace of the sequential execution. It is important to note that our approach does not explicitly construct an execution trace: we employ an online analysis that observes the events one by one, and updates the necessary state after each event.

Each statement in the trace can be associated with a *parallel timestamp* which shows the earliest time at which this statement could (hypothetically) be executed without violating any dependences. Intraprocedurally, all dependences are trivially satisfied because the timestamps for the statements executed by a method always grow monotonically. As mentioned earlier, we use a model of function-level parallelism where the code within a function/method executes sequentially, but several methods may execute in parallel. For example, events 1, 2, 6, 7, 11, 24, 25, 35, and 45 all correspond to the execution of `main`, and their corresponding parallel timestamps 1, 2, 3, 4, 5, 6, 16, 17, and 18 increase monotonically. This approach is in contrast with ILP-oriented analyses, where statements in the same method could be executed in parallel.

In the best-case model used to compute the available method-level parallelism, each callee executes in parallel with its caller. For example, the constructor invoked at trace event 2 runs in parallel with `main`. This is reflected by the timestamps associated with trace event 3 (the first statement executed in the constructor body) and trace event 6 (the first statement in `main` after the constructor call returns): both have a timestamp of 3 and thus will be executed in parallel. Of course, dependences through shared memory locations should be taken into account when computing the timestamps. For example, the call to `sub` (trace event 11) has a timestamp of 5, but the first statement in the called method (trace event 12) has a timestamp of 7. The reason is that `p1=c.y` reads the value of field `y` in the object *o2* pointed-to by `w2`. However, there is a write to that location at trace event 9, which has a timestamp of 6. Thus, the earliest possible timestamp for `p1=c.y` is 7. A similar situation is observed for events 35 and 36: even though the call to `sub2` has a timestamp of 17, statement `t1=e.x` cannot have timestamp 18 because it has a dependence on event 29 (`this.x=v1`) whose timestamp is 20.

Program code	Sequential trace (left) with parallel timestamps (right)	Field accesses

`class Vec{`
`float x, y;`
`Vec(float a, float b){`
`this.x = a;`
`this.y = b;`
`return;`
`}`
`static Vec sub(Vec c,`
`Vec d){`
`float p1, q1, r1;`
`float p2, q2, r2;`
`p1 = c.y;`
`q1 = d.y;`
`r1 = p1 - q1;`
`p2 = c.x;`
`q2 = d.x;`
`r2 = p2 - q2;`
`Vec s = new Vec;`
`s.Vec(r2,r1);`
`return s;`
`}`
`void sub2(Vec e,`
`Vec f){`
`float t1, u1, v1;`
`float t2, u2, v2;`
`t1 = e.x;`
`u1 = f.x;`
`v1 = t1 - u1;`
`this.x = v1;`
`t2 = e.y;`
`u2 = f.y;`
`v2 = t2 - u2;`
`this.y = v2;`
`return;`
`}`
`static void main(){`
`Vec w1 = new Vec;`
`w1.Vec(1.0,2.0);`
`Vec w2 = new Vec;`
`w2.Vec(3.0,4.0);`
`Vec w3 = sub(w2,w1);`
`Vec w4 = w1;`
`w3.sub2(w3,w4);`
`w3.sub2(w3,w2);`
`return;`
`}}`
	`1: w1 = new Vec // 1`
	`2: w1.Vec(1.0,2.0) // 2`
	`3: this.x = a // 3`
	`4: this.y = b // 4`
	`5: return // 5`
	`6: w2 = new Vec // 3`
	`7: w2.Vec(3.0,4.0) // 4`
	`8: this.x = a // 5`
	`9: this.y = b // 6`
	`10: return // 7`
	`11: w3 = sub(w2,w1) // 5`
	`12: p1 = c.y // 7`
	`13: q1 = d.y // 8`
	`14: r1 = p1 - q1 // 9`
	`15: p2 = c.x // 10`
	`16: q2 = d.x // 11`
	`17: r2 = p2 - q2 // 12`
	`18: Vec s = new Vec // 13`
	`19: s.Vec(r2,r1) // 14`
	`20: this.x = a // 15`
	`21: this.y = b // 16`
	`22: return // 17`
	`23: return s // 15`
	`24: w4 = w1 // 6`
	`25: w3.sub2(w3,w4) // 16`
	`26: t1 = e.x // 17`
	`27: u1 = f.x // 18`
	`28: v1 = t1 - u1 // 19`
	`29: this.x = v1 // 20`
	`30: t2 = e.y // 21`
	`31: u2 = f.y // 22`
	`32: v2 = t2 - u2 // 23`
	`33: this.y = v2 // 24`
	`34: return // 25`
	`35: w3.sub2(w3,w2) // 17`
	`36: t1 = e.x // 21`
	`37: u1 = f.x // 22`
	`38: v1 = t1 - u1 // 23`
	`39: this.x = v1 // 24`
	`40: t2 = e.y // 25`
	`41: u2 = f.y // 26`
	`42: v2 = t2 - u2 // 27`
	`43: this.y = v2 // 28`
	`44: return // 29`
	`45: return // 18`
		o1
		
		o1.x
		o1.y
		
		o2
		
		o2.x
		o2.y
		
		o2.y
		o1.y
		
		o2.x
		o1.x
		
		o3
		
		o3.x
		o3.y
		
		o3.x
		o1.x
		
		o3.x
		o3.y
		o1.y
		
		o3.y
		
		o3.x
		o2.x
		
		o3.x
		o3.y
		o2.y
		
		o3.y

Fig. 1. Sequential trace annotated with parallel timestamps and field read/write information.

At call sites where the callee returns a value, the caller continues execution until that return value is needed. For example, after `w3=sub(w2,w1)` (trace event 11 with timestamp 5), the next statement in `main` (trace event 24) has a timestamp 6 because it does not depend on `w3`. However, the following statement `w3.sub2(w3,w4)` (trace event 25) depends on the value of `w3` to obtain the receiver object of the call. This value becomes available only after trace event 23 (`return s` with timestamp 15); thus, the call to `sub2` has timestamp 16. This model of execution is consistent with the standard mechanism of *futures*, which in Java are available through `java.util.concurrent.Future`. Whenever a local variable is assigned the return value of a

call, subsequent reads of this variable have to occur later than the `return` statement that produced the value. These are the only dependences through local variables that need to be taken into account: due to the increasing timestamps assigned to the sequence of statements executed within a method, all other dependences through locals are trivially satisfied. (For the same reason, all dynamic control dependences from the sequential trace are also satisfied.)

The largest timestamp in the example is 29, while the number of executed statements is 45. The available parallelism is their ratio (about 1.6). One might be tempted to interpret this ratio as the speedup achievable by parallelizing the program. However, this idealized speedup will not be possible in practice due to limited hardware resources, overhead of thread creation and synchronization, complexity of code changes, etc. Instead, our work uses a different interpretation of the amount of available parallelism. We consider it to be a valuable measure of the parallelism inherent in the program, independent of the idiosyncrasies of hardware and the JVM. In particular, when this inherent parallelism is low, this is strong indication that there are fundamental problems for any attempts to parallelize the program "as is". For example, in one case study, the available parallelism was 1.8, which was disappointingly low. When we used the analysis to search for parallelism bottlenecks (as described later), we found an aspect of the computation that essentially serialized a large number of calls. This part of the code would have to be modified if there is any hope for a scalable parallel version. After a simple refactoring, the available parallelism increased to 102.7, indicating that it would likely be easier to construct a scalable parallel program based on this modified sequential program.

Program instrumentation. The analysis performs Java bytecode instrumentation before the program is executed. In this instrumentation, the following *shadow locations* are introduced. First, for every instance field (i.e., object field) *fld* declared in a given class, declarations for a write shadow instance field *fld_w* and a read shadow instance field *fld_r* are added to the class; both fields are of type `int`. For any run-time object o that is an instance of this class or any of its subclasses, $o.fld\_w$ will store the timestamp at which $o.fld$ was last written. The value of $o.fld\_r$ will store the largest timestamp among all reads of $o.fld$ that occurred since the last write to $o.fld$. Similarly, static shadow fields are introduced for each static field. Using these shadow fields, the analysis can track flow dependences (read-after-write), anti dependences (write-after-read), and output dependences (write-after-write) due to the corresponding fields in the program.

Next, each method is instrumented with shadow local variables, again of type `int`. As discussed earlier, only local variables that are assigned the return values of method calls—for example, $loc = m(\ldots)$—require corresponding shadow variables. Furthermore, only a write shadow local loc_w is needed for such a local loc, since only flow dependences through local variables need to be accounted for. During the subsequent dynamic analysis, loc_w will store the timestamp at which the return value of the called method becomes available. A special local variable $control$ is also introduced; as each statement in the method is executed, this variable is updated with the timestamp associated with this statement.

For each statement s in the method body, instrumentation statements are inserted immediately before s to update the shadow locations as necessary. The simplest case is when s does not read or write any location that has a corresponding shadow location. For such statements, the instrumentation increments the value of $control$. For more complex cases, details about the instrumentation are available in [26].

III. Parallelism-Inhibiting Dependences

The analysis described in the previous section computes a timestamp for each executed statement and then reports the ratio between the total number of statements and the largest timestamp ever computed. This measure of *available parallelism* cannot be expected to be actual speedup that is achievable by parallelization. Instead, our goal is to use such a hardware/JVM-neutral characterization as the basis for a number of other analyses. Later we describe case studies that demonstrate how these analyses uncover parallelism bottlenecks in Java programs.

Characterization of memory locations. Dependences created by accesses to memory locations could be significant bottlenecks for parallelism. Locations that would be shared among threads in a parallel execution—instance fields, static fields, and array elements—are of particular interest. A modified version of the dynamic analysis can be used to provide high-level characterization of the effects of each of these three categories. Specifically, all dependences due to locations from a particular category can be ignored. For example, consider statement `p1=c.y` from method `sub` in the running example. The instrumentation for this statement contains $control = 1 + max(c.y\_w, control)$. However, if we wanted to ignore all dependences through instance fields, this instrumentation would simply be $control = 1 + control$ and the computed timestamp would be 6 instead of 7. In our implementation, instead of re-instrumenting the code, a run-time flag is used to decide whether $c.y\_w$ should be ignored.

Of course, timestamps computed in this manner correspond to an invalid parallel execution, as certain dependences from the sequential trace will not be satisfied. As a result, the available parallelism will increase. The degree of this increase signifies the importance of dependences through this category of memory locations—in other words, how negatively they will impact parallel performance. In experiments with 26 programs we found that (1) with the exception of one program, static fields had negligible effect on parallelism, and (2) array elements had limited effect. On the other hand, instance fields had significant effect on the available parallelism.

Search for parallelism bottlenecks. Dependences through instance fields can create significant parallelism bottlenecks. Given a program with many fields, our goal is to automatically discover a (hopefully small) subset of problematic fields. One version of this approach considers the effects of ignoring all dependences through $o.fld$ (in all run-time objects o) for a

particular field *fld*—informally, "turning off" the field—while respecting all other dependences. This is done independently for each instance field *fld* in the program, through a separate run of the analysis for this particular *fld*. The increase in available parallelism is used for ranking. A field that causes a higher increase is more likely to be a source of parallelism-inhibiting dependences, and should be among the first that a programmer inspects.

There could be several fields that, independently from each other, create parallelism bottlenecks. Turning off one of them will not increase the available parallelism, due to the effects of the remaining problematic fields. In this case, a second version of the approach is used. The analysis is first executed with a baseline configuration in which all instance fields are turned off. Then, for each instance field *fld*, a configuration is executed with *fld* turned on and the rest of the fields still turned off. The available parallelism with this configuration will be lower than with the baseline one. The difference between the two is an indication of the significance of dependences through *fld*. The results of all runs are used to rank the fields: the higher the difference observed for the *fld*-specific run, the higher the rank of *fld*.

For both approaches, our implementation does not re-instrument the code for each run, but rather employs instrumentation-time-generated field IDs to specify which dependences to ignore during a run.

IV. Experimental Evaluation

The analyses described earlier were implemented with the help of the Soot transformation framework [25]. All bytecode in program classes and library classes is instrumented (including the standard Java libraries). In addition to the values of the relevant shadow locations, the run-time events produced by the instrumentation contain information about the category of statement being observed (e.g., read of an instance field, together with a unique field ID).

The evaluation was based on 26 Java programs from four benchmark suites: SPEC JVM98, Java Grande v2.0 (section 3) [11], a Java version [15] of the Olden benchmarks, and DaCapo 2006-MR2 [5]. Only single-threaded applications were used, since the analysis is designed to work on sequential programs. Table I describes the programs, with number of methods and instance fields. The grouping corresponds to the benchmark suite, in the order listed above.

All experiments were performed on a Dell PowerEdge R300 with a 2.83GHz Intel Xeon and 8Gb of RAM. The execution environment was running Linux 2.6.18 and a Java HotSpot 64-bit Server VM, build 1.6.0_11-b03, with 6Gb of heap memory. The Java libraries instrumented and used were from this 1.6.0 release for Linux. Column 2 in the table shows the number of Soot IR statements observed in the program execution (i.e., the numerator in the computation of the available parallelism), including all executed library code. Due to non-deterministic JVM behavior such as the order of class loading and initialization through static initializer methods, and finalizer invocations due to garbage collection, the number of statements could differ slightly from run to run; in practice, any variations we observed were negligible.

Additional details on the benchmarks and the experiments are available in [26].

Analysis cost. Columns 3 and 4 describe the running time of the instrumented program and the overhead of the instrumentation. The median slowdown is a factor of 28.9, and 15 of the 26 benchmarks have a slowdown less than a factor of 30. Existing work which uses fine-grain instrumentation for Java exhibits overheads similar to ours—for example, the work in [32] shows overhead of about $37\times$. We have not yet attempted to introduce any sophisticated optimizations of the instrumentation. The only optimization employed is a straightforward merging of consecutive $control = 1 + control$ statements into a single update. Our current work focuses on using the results of the analysis to uncover parallelism bottlenecks; reducing the cost of gathering these results is the target of future work. Similar considerations apply for the memory overhead (not shown in the table due to space limitations). Without any attempts at optimizations to reduce memory overhead, the median increase in memory usage was a factor of 6.8, with 17 of the benchmarks showing a memory overhead of less than a factor of 10.

Available parallelism. Column 5 shows the measurements of available parallelism, obtained as described in Section II. In general, the observed best-case parallelism is low (with a few exceptions): the median idealized speedup is 2.1. These results have two implications. First, if one were to consider ways to execute the program "as is" (e.g., through speculation, where each callee executes speculatively in parallel with its caller), the potential for improvements is rather limited. Second, the importance of identifying and removing parallelism bottlenecks is clear: either through manual programmer effort or through automatic transformations, the programs will have to be re-engineered if there is any hope of achieving scalable parallel performance.

Characterization of categories of memory locations. Data dependences through memory locations that would be shared among threads—instance fields, static fields, and array elements—are potential bottlenecks for parallelism. Column 6 in Table I provides characterization of each of these three categories. All data dependences through locations from a category are ignored during the dynamic analysis, which increases the measured available parallelism. The degree of this increase is an indication of the importance of these locations in a parallelization effort.

Reads and writes of static fields have very limited effect on the available parallelism, and do not appear to create problems for manual re-engineering and parallelization. One notable exception is `moldyn`; the next section presents a case study with this program. Dependences through array elements also do not seem to present a significant problem for parallelism. Ignoring the dependences through instance fields leads to significant increase in available parallelism. Thus, programmer efforts (and the supporting software tools) should pay special attention to modeling and understanding the shared state stored

(1) Program			(2) #Stmt	(3) Time	(4) Slowdown	(5) Available	(6) Parallelism when ignoring			(7) Uninstrumented libs	
Name	#Mth	#InsFld	(million)	(mm:ss)	(×)	parallelism	stat. flds	inst. flds	arrays	av. paral.	Δ time
compress	164	52	612	0:38.6	91.7	1.2	1.2	3.3	1.2	1.2	59%
db	164	23	1062	1:50.5	20.0	1.2	1.2	25.7	1.2	1.1	77%
jack	445	131	33	0:06.3	10.3	1.7	1.8	2.1	1.8	1.2	79%
javac	1300	282	38	0:05.7	7.6	2.7	3.1	21.0	2.7	2.7	63%
jess	804	215	909	2:24.0	97.4	5.2	5.2	56.7	5.2	5.5	52%
mpegaudio	441	179	56	0:35.9	71.7	1.6	1.6	2.1	1.9	1.6	87%
raytrace	294	89	1249	1:34.7	99.6	2.4	2.4	121.8	2.4	2.4	47%
euler	31	55	3158	15:24.5	653.2	1.6	1.6	4.0	1.8	1.6	56%
moldyn	23	66	724	0:15.3	29.6	1.8	118.1	1.8	1.8	1.8	0%
montecarlo	182	70	1699	1:01.8	20.2	1.3	1.3	298.4	2.7	1.5	48%
JGraytracer	73	51	1753	0:25.3	44.0	1.2	1.2	1.3	1.2	1.2	11%
search	32	17	3388	8:20.9	241.8	1.6	1.6	1.6	1.6	1.6	48%
bh	63	17	1608	5:42.9	222.8	47.4	47.4	47.7	47.5	52.0	35%
bisort	17	3	238	0:04.6	14.8	1.1	1.1	1.1	1.1	1.1	3%
em3d	29	8	1382	1:19.7	102.2	4.0	4.0	5134.0	4.0	4.1	44%
health	21	20	935	0:29.1	19.5	116.9	116.9	125.9	116.9	47.2	72%
power	33	26	595	1:04.3	131.6	134.1	134.1	14168.5	173.6	134.8	60%
tsp	17	7	2128	0:25.5	14.2	38.2	38.2	40.8	38.2	36.5	7%
voronoi	67	12	1084	1:22.9	45.7	32.1	32.1	33.4	32.1	32.1	32%
antlr	2683	632	122	0:07.8	11.7	1.6	1.6	3.7	1.7	1.4	56%
bloat	4260	1089	1744	1:26.2	28.3	5.1	5.1	18.4	6.1	20.3	76%
chart	13448	3638	506	0:46.7	19.5	2.0	2.0	5.9	2.0	36.0	78%
fop	26135	7478	397	0:38.9	16.6	2.4	2.4	4.5	2.4	1.9	82%
jython	13881	3489	541	0:29.4	17.2	2.2	2.2	2.5	2.2	1.4	56%
luindex	2740	1036	171	0:28.1	29.6	4.0	4.0	7.3	4.0	6.1	67%
pmd	14391	3561	14	0:03.4	4.3	2.0	2.0	4.1	2.0	4.4	57%

TABLE I
EXPERIMENTAL STUDIES.

in instance fields.

Ignoring the standard libraries. All results presented until now were obtained with instrumentation of both program code and code from the standard Java libraries. A possible alternative is to instrument only the program code. Column 7 describes the results of this experiment. The second sub-column shows the reduction in analysis running time, relative to the measurements in column 3. Clearly, significant savings can be obtained in this manner: the median reduction is 56%. However, the measurements of available parallelism could also be affected when the effects of library code are not taken into account. The difference between column 5 and the first sub-column of column 7 is significant for several programs. While future investigations are needed to provide more insights into these results, our preliminary conclusion is that measurements of available parallelism should be performed with instrumented standard libraries.

V. CASE STUDIES

Case study: moldyn. Only for `moldyn` we see that dependences through static fields are likely to affect the parallelism; as a result, it was chosen for our first case study. The program contains 12 static fields, seven of which are final (read-only); we focused on the remaining five fields. Three fields immediately stand out: `epot`, `vir`, and `interactions`, all of them updated in method `force` from class `particle`. For example, the update of `epot` is `epot = epot + expr`. The updates are specific to this invocation of `force` and can be computed independently of any other invocations of the method. The updates of the other two static fields follow the same pattern.

Clearly, the updates to these fields do not allow multiple invocations of `force` to run in parallel with each other, as each invocation has to wait for the previous one to perform the field updates. Eliminating this bottleneck is straightforward. We created an array with number of elements equal to the number of times `force` is called; each array element is a triple of values, one per static field. Within a single invocation, all updates are performed on the invocation-specific array elements. After all invocations are completed, the array elements are added up. This new sequential program is equivalent to the original one (assuming associativity of addition operations), but its measured available parallelism increases from 1.8 to 102.7. This value is close to the value of 118.1 measured when completely ignoring static fields (column 6 in Table I). This indicates that the changes have eliminated the problem. In general, after such a transformed sequential program is created, it is easy and desirable to run regression tests against the original program. This way correctness testing can start early, even before actual parallelization is attempted.

Case study: em3d. In column 6 of Table I, program `em3d` shows the greatest relative increase in the available parallelism when dependences through all instance fields are ignored. As a result, the program was chosen for our second case study. Using the approach from Section III, we investigated each instance field by ignoring dependences through this field

(while respecting all other dependences) and measuring the resulting available parallelism. This was done independently for each instance field declared in program classes. Only field `fromCount` in class `ENode` caused any significant changes, with available parallelism increasing from 4.0 to 56.4.

The program creates and manipulates a bipartite graph with nodes represented by `ENode` instances. A node has a field `toNodes`, which refers to an array in which elements are references to the successor nodes in the graph. A similar field `fromNodes` provides access to the predecessor nodes. Method `makeUniqueNeighbors`, when invoked on an `ENode` object o_1, populates its `toNodes` array. During this, when a reference to an `ENode` object o_2 is added to o_1's `toNodes`, field `fromCount` in o_2 is incremented. After all calls to `makeUniqueNeighbors` complete, the `fromCount` values (i.e., numbers of predecessors) are used to create the predecessor lists `fromNodes`.

The inspection of `makeUniqueNeighbors` and how it manipulates `fromCount` took us only a few minutes and immediately highlighted the problem: calls to this method on different nodes are not independent from each other. Calls on two nodes o_1 and o_3, during which the same node o_2 is added as a successor of both, will result in conflicting increments of o_2's field `fromCount`. The solution is clear: remove the updates to `fromCount` from the method, in order to allow all calls to `makeUniqueNeighbors` to be independent of each other. After these calls are completed, all `toNodes` arrays are traversed to compute all `fromCount` values. After this change, the measured available parallelism jumps from 4.0 to 48.9.

This study provides an example of interleaving of a computation that is highly parallel (populating the successor arrays) with a computation that exhibits many dependences (computing the number of predecessors). The programmer of a sequential program will naturally interleave these computations, because the code is simple and clean when written in this manner. But this obvious choice becomes a major bottleneck for parallelism. The untangling of the two is a minor algorithmic change that can be easily done by a programmer once the problematic field is identified. Note that automatic code transformation is essentially impossible in this case, since correctness depends on high-level properties of the graph-creation algorithm.

Case study: raytracer. Java Grande `raytracer` shows very little available parallelism. This is surprising because, at the algorithmic level, raytracing is inherently parallel. A raytracer renders a two-dimensional canvas from a three-dimensional scene, and the color of each canvas pixel can be computed independently from the colors of other pixels. The low available parallelism was the reason to choose this program for the third case study.

The implementation of `raytracer` renders all pixels through a single method call. We modified the code to partition the canvas into n sections and to make a separate rendering call for each section (all results are presented with $n = 32$). The hope was that the rendering calls would be independent of each other, leading to an increase in available parallelism. However, the available parallelism remained unchanged.

We investigated each instance field by ignoring dependences through this field (and respecting all other dependences). Among the more than fifty instance fields in the program, not a single one produced an increase in available parallelism. As described in Section III, in such a case a different approach can be applied: the analysis is executed with (1) all instance fields turned off, and (2) all instance fields except for one field *fld* turned off. With this approach, the decrease in available parallelism was insignificant for most fields *fld*. However, for several fields the decrease was indeed substantial. Four of them belong to class `Isect`; they are the only fields in the class.

We investigated `Isect` and found that only two places in the code create instances of this class. One of these initializes an instance field of type `Isect` in class `RayTracer`, with a comment describing the field as the "current intersection instance" and stating that "only one is needed". The manipulation of this `Isect` object occurs inside method `intersect`, where the four fields are read and written many times. Over multiple calls to `intersect`, there are conflicting updates of these four fields. It appears that the programmer's intent was to avoid creating multiple `Isect` objects (and the associated allocation and GC costs), and instead to have a single object that is reused. This is a significant bottleneck to parallelism. We introduced `Isect` objects as necessary, in order to avoid the conflicts among calls to `intersect`.

This code change did not increase the available parallelism. We recomputed the ranking of instance fields for the modified program, and observed that the fields in `Isect` were no longer indicated as problematic, meaning that our changes indeed removed the dependences through these fields. Among the problematic fields, a natural next target was a `checksum` field in `RayTracer`. For each pixel, just after that pixel's color is calculated, the color is added to this field. These updates create dependences between several rendering calls. We moved the checksum computation just after all of the image has been processed.

This change did not increase the available parallelism, but `checksum` was not reported anymore as a problematic field in the recomputed rankings. The next target were the two (and only) fields from class `Ray`, both of which were reported as bottlenecks. Only two locations in the code create `Ray` objects, and one of them is an initialization of a field `tRay` in `RayTracer`, with a code comment "temporary ray". The use of `tRay` in a sequence of calls made by method `render` inadvertently serializes the multiple calls to `render`. We changed `tRay` from a field to a local variable of `render` with the appropriate initialization with a new `Ray` object, and added a corresponding parameter to methods called by `render`.

This change eliminated the fields of `Ray` from the recomputed list of parallelism-inhibiting fields. The only highly-ranked fields in this list, after all changes described above, were the three (and only) fields in class `Vec`. We focused on all instance fields of type `Vec`. One of them was a field `L` in `RayTracer`, described by a comment as a "temporary

vector". Inspecting the usage of `L`, it does in fact appear to be temporary in the sense that its value is set in the scope of a single method call, never used again after that call returns, and not altered by any other methods during its usage. By changing it to a local variable in the method that uses it, this bottleneck was removed.

Another field of type `Vec` that stands out is `Sphere.v`, which is initialized with a `Vec` object and never changed. The code comment is "temporary vector used to minimize the memory load". The updates to the object's fields occur in method `intersect` in class `Sphere`. The field updates are through calls to a method `sub2` similar to the one shown in Figure 1. These updates serialize multiple calls to `intersect`. Making `v` a local variable of `intersect` (initialized with a fresh `Vec` object) eliminated the problem.

After these five independent sources of parallelism-inhibiting dependences were discovered and eliminated, the available parallelism finally increased from 1.2 to 21.4. Note that *all* of these bottlenecks need to be removed—omitting even a single one will keep the available parallelism at the original level.

VI. Related Work

A number of approaches have been proposed to measure the available parallelism at the statement (or instruction) level (e.g., [1], [12], [14], [16], [20], [22], [28]). Typically, in these and similar studies, a program execution trace is first created; next, possible parallel schedules are defined by taking into account the dependences between binary instructions in the trace, under various assumptions. A notable exception is Kumar's work [12], which does not create a trace but instead instruments the program to compute the parallel schedule online. We adopt a similar approach by defining an online analysis; this avoids the cost of storing an explicit trace. However, our target is parallelism at the level of Java methods, not at the level of individual bytecode instructions. Researchers have also considered dynamic analysis of loop-level parallelism, where iterations of a loop may run concurrently with each other [2], [13], [17], [21], [23], [24], [31], [33].

Recent work in [9] measures the available instruction-level parallelism in Java. A run of the program produces a trace, which is later analyzed to construct a dynamic dependence graph and to compute timestamps for instruction-level parallelism, similarly to approaches such as [1]. These timestamps are used to identify loops that may be possible and profitable to parallelize. In contrast, our approach considers method-level parallelism, does not create an explicit dependence graph, and finds potential bottlenecks by turning off certain dependences.

Larger units of parallelism have also been considered. In [27], user-specified tasks are analyzed for dynamic dependences. The approach is based on sampling and does not uncover all run-time dependences. In [18], a study of speculative thread-level parallelism considers several combinations of parallelization for loops and/or procedure calls, and evaluates their potential speedups on a given binary execution trace. A similar speculation-based approach is proposed in [29], with additional modeling of thread-management overhead. Speculation is also the target of the work in [19], where a software-based speculative multithreading architecture is implemented in a Java virtual machine and can be used for dynamic analysis of speculative method-level parallelism (e.g., to investigate the effects of speculation overhead and return value prediction). Earlier work [3], [4], [10] proposes hardware and software for similar speculative Java execution, and performs simulation studies on it. None of these approaches have been used to identify specific parallelism bottlenecks in the program, in order to point a programmer toward necessary code changes.

VII. Conclusion

We believe that automated tool support will be critical for uncovering bottlenecks to parallelism in sequential programs. Such problems may be the unintended consequences of choices made by the programmer when the sequential program was first created. We describe a dynamic analysis of available parallelism in Java programs which can be used as basis for such tool support. Measurements on 26 subjects reveal that the available parallelism, in general, is rather low. This likely means that these programs will not be easy to transform into scalable parallel versions, and also that speculative parallelism will not lead to significant performance improvements. In particular, dependences through fields of heap objects appear to be a significant impediment to parallelism. To address this problem, we propose an automated technique to identify parallelism-inhibiting fields. With the help of this analysis, the problematic code was discovered in the course of our case studies. This experience indicates that programmer productivity could be increased by effective tools that assist with step-by-step identification and removal of parallelism bottlenecks in sequential Java programs.

Acknowledgments. We thank the ICSM reviewers for their valuable and thorough comments, and Mark Marron for providing a Java version of the Olden benchmarks. This research was supported in part by the National Science Foundation under grants CCF-0546040 and CCF-0811781.

References

[1] T. Austin and G. Sohi, "Dynamic dependency analysis of ordinary programs," in *International Symposium on Computer Architecture*, 1992.

[2] M. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. August, "Revisiting the sequential programming model for multi-core," in *International Symposium on Microarchitecture*, 2007.

[3] M. Chen and K. Olukotun, "Exploiting method-level parallelism in single-threaded Java programs," in *International Conference on Parallel Architectures and Compilation Techniques*, 1998.

[4] ——, "The Jrpm system for dynamically parallelizing Java programs," in *International Symposium on Computer Architecture*, 2003.

[5] "DaCapo benchmarks," `www.dacapobench.org`.

[6] D. Dig, J. Marrero, and M. Ernst, "Refactoring sequential Java code for concurrency via concurrent libraries," in *International Conference on Software Engineering*, 2009.

[7] M. Fähndrich, D. Garbervetsky, and W. Schulte, "A static analysis to detect re-entrancy in object oriented programs," *Journal of Object Technology*, vol. 7, no. 5, Jun. 2008.

[8] R. Ghiya, L. Hendren, and Y. Zhu, "Detecting parallelism in C programs with recursive data structures," in *International Conference on Compiler Construction*, 1998.

[9] C. Hammacher, K. Streit, S. Hack, and A. Zeller, "Profiling Java programs for parallelism," in *International Workshop on Multicore Software Engineering*, 2009.

[10] S. Hu, R. Bhargava, and L. K. John, "The role of return value prediction in exploiting speculative method-level parallelism," *Journal of Instruction-Level Parallelism*, vol. 5, no. 1, Nov. 2003.

[11] `www.epcc.ed.ac.uk/research/java-grande`.

[12] M. Kumar, "Measuring parallelism in computation-intensive scientific/engineering applications," *IEEE Transactions on Computers*, vol. 37, no. 9, Sep. 1988.

[13] J. Larus, "Loop-level parallelism in numeric and symbolic programs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 1, Jul. 1993.

[14] J. Mak and A. Mycroft, "Limits of parallelism using dynamic dependency graphs," in *International Workshop on Dynamic Analysis*, 2009.

[15] M. Marron, M. Mendez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur, "Sharing analysis of arrays, collections, and recursive structures," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Egineering*, 2008.

[16] A. Nicolau and J. Fisher, "Measuring the parallelism available for very long instruction word architectures," *IEEE Transactions on Computers*, vol. 33, no. 11, 1984.

[17] C. Oancea and A. Mycroft, "Set-congruence dynamic analysis for thread-level speculation (TLS)," in *International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[18] J. Oplinger, D. Heine, and M. Lam, "In search of speculative thread-level parallelism," in *International Conference on Parallel Architectures and Compilation Techniques*, 1999.

[19] C. Pickett and C. Verbrugge, "SableSpMT: A software framework for analysing speculative multithreading in Java," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Egineering*, 2006.

[20] M. Postiff, D. Greene, G. Tyson, and T. Mudge, "The limits of instruction level parallelism in SPEC95 applications," *SIGARCH Computer Architecture News*, vol. 27, no. 1, 1999.

[21] L. Rauchwerger and D. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1995.

[22] D. Stefanović and M. Martonosi, "Limits and graph structure of available instruction-level parallelism," in *Euro-Par: European Conference on Parallel Computing*, 2000.

[23] C. Tian, M. Feng, V. Nagarajan, and R. Gupta, "Copy or discard execution model for speculative parallelization on multicores," in *International Symposium on Microarchitecture*, 2008.

[24] G. Tournavitis, Z. Wang, Zheng, B. Franke, and M. O'Boyle, "Towards a holistic approach to auto-parallelization," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[25] R. Vallée-Rai, E. Gagnon, L. Hendren, P. Lam, P. Pominville, and V. Sundaresan, "Optimizing Java bytecode using the Soot framework: Is it feasible?" in *International Conference on Compiler Construction*, 2000.

[26] K. Van Valkenburgh, "Measuring and improving the potential parallelism of sequential Java programs," Master's thesis, Ohio State University, Aug. 2009.

[27] C. von Praun, R. Bordawekar, and C. Caşcaval, "Modeling optimistic concurrency using quantitative dependence analysis," in *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2008.

[28] D. Wall, "Limits of instruction-level parallelism," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[29] F. Warg and P. Stenström, "Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms," in *International Conference on Parallel Architectures and Compilation Techniques*, 2001.

[30] J. Wloka, M. Sridharan, and F. Tip, "Refactoring for reentrancy," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2009.

[31] P. Wu, A. Kejariwal, and C. Caşcaval, "Compiler-driven dependence profiling to guide program parallelization," in *International Workshop on Languages and Compilers for Parallel Computing*, 2008.

[32] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky, "Go with the flow: Profiling copies to find runtime bloat," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[33] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke, "Uncovering hidden loop level parallelism in sequential applications," in *IEEE International Symposium on High-Performance Computer Architecture*, 2008.