Evaluating the Effectiveness of Slicing for Model Reduction of Concurrent Object-Oriented Programs

Matthew B. Dwyer[1], John Hatcliff[2], Matthew Hoosier[2], Venkatesh Ranganath[2], Robby[2], and Todd Wallentine[2]

[1] University of Nebraska, Lincoln, NE 68588, USA
dwyer@cse.unl.edu
[2] Kansas State University, Manhattan, KS 66506, USA
{hatcliff, matt, rvprasad, robby, tcw}@cis.ksu.edu

Abstract. Model checking techniques have proven effective for checking a number of non-trivial concurrent object-oriented software systems. However, due to the high computational and memory costs, a variety of model reduction techniques are needed to overcome current limitations on applicability and scalability. Conventional wisdom holds that static program slicing can be an effective model reduction technique, yet anecdotal evidence is mixed, and there has been no work that has systematically studied the costs/benefits of slicing for model reduction in the context of model checking source code for realistic systems.

In this paper, we present an overview of the sophisticated Indus program slicer that is capable of handling full Java and is readily applicable to interesting off-the-shelf concurrent Java programs. Using the Indus program slicer as part of the next generation of the Bandera model checking framework, we experimentally demonstrate significant benefits from using slicing as a fully automatic model reduction technique. Our experimental results consider a number of Java systems with varying structural properties, the effects of combining slicing with other well-known model reduction techniques such as partial order reductions, and the effects of slicing for different classes of properties. Our conclusions are that slicing concurrent object-oriented source code provides significant reductions that are orthogonal to a number of other reduction techniques, and that slicing should always be applied due to its automation and low computational costs.

1 Introduction

1.1 Motivation

Model checking techniques have proven effective for debugging a number of non-trivial software systems. Due to the high computational and memory costs, a variety of model reduction techniques such as data abstraction [11], predicate abstraction [2], systematic application of data and resource bounds, heuristic search strategies [16], and partial order reduction techniques driven by synchronization and heap-structure properties [36, 12] are needed to overcome current limitations on applicability and scalability. In model reduction, the resulting transition system should be small enough to make automatic checking tractable, yet it should be large enough to capture all information relevant to the property being checked. One of the primary difficulties is determining which parts of the program are relevant to the property being checked.

H. Hermanns and J. Palsberg (Eds.): TACAS 2006, LNCS 3920, pp. 73–89, 2006.

Conventional wisdom holds that static program slicing can be an effective model reduction technique for software model checking. Given a program and a *slicing criterion* – a set of program points (e.g., statements) that a user is interested in, a program slicer automatically calculates the portions of the program that are relevant for carrying out the computation at the statements given in the criterion. All features of the program (e.g., statements, fields, classes, methods, threads) that are irrelevant for the computation at the criterion statements are "sliced away." Thus, by including the features of the program mentioned in a property/specification to be model checked as a slicing criterion, slicing will remove from the program features that are irrelevant for (i.e., do not influence) the property to be checked.

However, existing experience with slicing for model reduction is sometime inconclusive. For example, Holzmann's experience shows that slicing in Spin usually does not yield much reduction for realistic Promela design models [20]. While there have been extensive experimental studies to evaluate the effectiveness of model reduction techniques such as partial order reductions [15, 36, 12, 14] and abstraction [2] for software model checking, there have been no such studies to evaluate the effectiveness of slicing as a model reduction technique. In fact, there seem to be several factors that prevent researchers and practitioners from drawing strong conclusions about the effectiveness of slicing as a model reduction technique.

First, due to the lack of robustness of many model checking tools and the relative immaturity of the field in general, researchers often tried to evaluate the effectiveness of slicing using only small "text book" examples such as Dining Philosophers or Bakery Algorithm in which the system under consideration has already been boiled down to its bare essentials and possesses no property-irrelevant information to be sliced away. In contrast, larger realistic systems often have many aspects that are irrelevant to specific properties being checked.

Second, few software model checking frameworks include or make use of existing program slicing frameworks (quite sophisticated static analysis frameworks themselves – especially in the context of concurrent object-oriented programming languages). The Bandera model checking framework was the first to include slicing capabilities with Spin following shortly thereafter with a Promela slicing capability. Although one could imagine applying existing C program slicers in the context of model checking C programs, we know of no other software model checking frameworks with integrated slicing capabilities.

Lack of available slicing infrastructure in general has further hindered progress in evaluating the effectiveness of slicing for model reduction. Simply put, building a slicing framework capable of scaling to realistic applications and handling challenging language features such as one would find, e.g., in full Java, is a very challenging task.

1.2 This Paper

In this paper, we present an overview of our Java program slicer available as part of Indus, a program analysis tool kit, and we use this slicer to carry out wide-ranging experimental studies that demonstrate program slicing to be a valuable model reduction technique for model-checking concurrent object-oriented software. Both the slicer and our extensible model checking framework Bogor [33] are components of the next generation of the Bandera Java model checking framework that translates Java programs into models in the Bandera Intermediate Representation (BIR). Bogor model checks BIR models, using state of the art techniques for heap symmetry, collapse compression for object-oriented data structures, and partial order reductions driven by synchronization and escape analysis.

We believe this work presents a number of results that will be of interest to both researchers and practitioners working in the area of software model checking.

- It represents the first study to systematically demonstrate the effectiveness of program slicing as a model reduction technique for an interesting range of Java programs. Specifically, we consider ten different programs including an example from the Java Grande benchmarks as well as programs based on Siena – a generic scalable publish/subscribe Internet event-notification framework.
- It considers the relative benefits of state-of-the-art Java slicing techniques with state-of-the-art implementations of other model reduction techniques such as partial order reductions, and demonstrates that reductions provided by slicing are largely orthogonal to the effect of these other techniques.
- It considers the effects of slicing for model reduction when considering different classes of properties including deadlock checking and assertions.
- It shows that slicing can be applied as a low-cost (the costs are a very small percentage of the overall cost of model checking) and completely automated model reduction technique that almost always yields both space and time reductions (often, significant reductions) while almost never causing an increase in end-to-end run time.
- The Indus tool kit provides a sophisticated program slicer capable of handling full Java and scaling to interesting off-the-shelf concurrent Java programs of more than 10,000 application bytecodes (49,314 application + library bytecodes). Indus is freely available, and has been downloaded over 1100 times since its public release in June 2004. Thus, the techniques described in this paper can be *immediately* applied to other Java model checking frameworks such as JPF [5] and jMoped [37]. In fact, Indus is already being used by researchers at Fujitsu for reduction when model checking with JPF.

The rest of the paper is organized as follows. Section 2 overviews the architecture of the Bandera Java model checking tool set. Section 3 gives a brief summary of slicing and the Indus analysis framework. Section 4 explains how slicing can be used for model reduction, and presents the hypotheses and research questions that we seek to answer with our experimental studies. Section 5 provides an overview of the different examples that we consider along with metrics capturing various static characteristics of their implementation (no. of bytecodes, no. of classes, no. of fields, etc.). Section 6 presents the results of our experiments and provides assessment with respect to the previously given research hypotheses. Section 7 surveys related work and Section 8 concludes.

The Indus web site [29] provides the Indus distribution as well as an extended version of this paper that gives an expanded discussion of experimental results.

2 Bandera Overview

Bandera is a tool framework for model checking concurrent Java programs. The tool framework is organized as a *modular pipeline* – each tool in the pipeline communicates with its predecessor and successor purely through its inputs and outputs. To create a run of Bandera for a specific program and property, the user creates a *session file* that indicates the components of the pipeline that should be applied as well as particular settings or options for each component.

In the pipeline that we use for the experiments in this paper, the **Soot** tool [39] first reads Java class files to be model checked and translates them to Jimple [39] – Soot's three-address code intermediate representation for Java. The **Indus** tool takes the resulting Jimple program along with information about the property to be checked and produces a sliced Jimple program. The **Jimple-To-BIR (J2B)** tool takes the sliced Jimple program and translates it to BIR, which is then model checked by **Bogor**.

We are interested in determining the extent to which slicing can improve performance *beyond* the best reduction strategies currently available in Bogor.

These reduction strategies include heap symmetry/canonicalization which represents all execution states of a Java program that differ only in the physical addresses of objects or in the unreclaimed garbage using a single representative state, thread symmetry, and collapse compression which reduces the space required to store a state by sharing common parts of distinct states [34].

In this paper, we will be most interested in Bogor's partial order reduction framework that minimizes the set of paths that need to be explored in the state-space during model checking. Classical partial order reductions (POR) (e.g., [15]) leverage the *independence* of transitions to induce equivalence classes of paths such that it is sufficient to explore a single path from each class. For multi-threaded Java programs, we leverage the structure of the Java heap and Java synchronization idioms to infer more precise information about transition independence [12]. For example, transitions that access *thread-local* objects (i.e., objects that are reachable from a single thread) are independent because no transition in another thread can possibly access such an object (until the object becomes shared). Transitions that operate on a *properly locked* object (e.g, where the set of locks held by each thread when accessing the object always contains at least one common lock) are also independent as are operations on *read-only* objects. An additional feature of Bogor's POR implementation is that it biases the search to coalesce transitions in a method into a consecutive run of transitions. By doing this, the model checker is able to defer state-storage until the end of the run of independent transitions effectively implementing on-the-fly detection of *atomic blocks* of transitions. The combination of partial order reduction techniques in Bogor yields orders of magnitude reduction in the space and time required for model checking nearly all of the Java programs we have encountered.

3 Program Slicing and the Indus Java Slicing Framework

3.1 A Brief Overview of Slicing Concurrent Java Programs

There are many variants of slicing (forward/backward, static/dynamic) [38]. We consider *static backward* slicing – the variant usually applied for model reduction [6, 18, 26].

Static backward slicing uses *static* program analysis to look *backwards* along data and control flows to discover the set S_C of all program statements that influence the given slicing criterion C – a set of select program statements. S_C is guaranteed to contain all the statements upon which computations at statements in C depend; S_C will usually contain additional program statement beyond those that actually influence C due to the conservative nature of static analysis.

Slicing for concurrent Java programs is based on several notions of program statement dependence, and it is well beyond the scope of this paper to present detailed definitions for each of these. Our previous work presents formalizations and correctness proofs for the definitions of data and control dependence [18, 31] and dependences for concurrent Java [17] that we use in this paper. Control and data dependences are well-understood, so we focus below on the less familiar, but crucial, notions of dependence needed to treat concurrency. Briefly, a node n is data-dependent on node m if, for a variable v referenced at n, a definition of (*i.e.*, an assignment to) v at m reaches n; a node n is *control-dependent* on a conditional statement m if one of the branches of m must always lead to n but the other branch of m can bypass n and reach the program's (i.e., method's) end node.

For the definitions below, if a statement at control-flow graph (CFG) node n is in the slice set S_C and n depends on a statement at node m, then node m is also in S_C. Intuitively, the slicer begins with the statements in the slicing criteria C and computes a transitive closure using data and control dependence along with the dependences below (modulo some optimizations enabled by further static analysis) to obtain S_C.

Divergence: To preserve behaviors with *infinite delay* (e.g., as required for checking liveness properties or certain classes of deadlocks), our earlier work [18, 31] notes that it is important to consider additional notions of control dependence that preserve diverging executions (e.g., as caused by infinite loops). One method for capturing dependences caused by diverging loops is to use the notion of *strong post-domination* and *weak control dependence* introduced by Podgurski and Clarke [28]. Node n *strongly post-dominates* node m if n post-dominates m and there is an integer $k \geq 1$ such that every path from node m of length $\geq k$ passes through n [28]. The difference between strong post-domination and the simple definition of post-domination above is that even though node n occurs on every path from m to e (and thus n post-dominates m), it may be the case that n does not strongly post-dominate m due to a loop in the CFG between m and n that admits an infinite path beginning at m and not containing n. Hence, strong post-domination is sensitive to the possibility of non-termination along paths from m to n. Node n_j is *weakly control dependent* on n_i if n_i has at least two successors, n_k and n_l, n_j strongly postdominates n_k, n_j does not strongly postdominate n_l. We will typically refer to control dependence and weak control dependence as *termination insensitive* and *termination sensitive* control dependence, respectively. The experiments in this paper will use the notion of weak control dependence – which guarantees that if n is in the slice and there exists a (possibly infinite) loop in the CFG that could prevent control from reaching n, then the control structure associated with the loop is also included in the slice.

Interference dependence: To capture data dependence across threads, node n from thread t_n is *interference dependent* on node m from thread t_m with $t_n \neq t_m$ if there exists a variable $v \in$ *def*$(m) \cap$ *ref*(n) and there exists a schedule in which m's execution is followed by n's execution with no intervening definition of m [25]. A precise *static* calculation of schedules that may give rise interference dependence is prohibitively expensive for large systems (exact precision is equivalent to the model checking problem itself). Cheap but conservative/imprecise strategies [17, 26] simply assume that an interference exists whenever $v \in$ *def*$(m) \cap$ *ref*(n) even though there may never be a schedule that gives rise a flow of data between m and n. Other approaches use exponential symbolic execution algorithms to try to detect realizable execution paths leading to such flows [25, 27]. Indus strikes a balance between these by pruning infeasible interference edges using escape/alias analysis and various "happens before" relations [32].

Ready dependence: Just as diverging loops may give rise to infinite delays within a single thread, indefinite delays can also be generated due to interactions between threads via synchronization primitives such as locking (synchronized statements) and wait/notify. When the completion of a statement n of thread t_n (e.g., an lock acquire or a wait) depends of the completion of node m of thread t_m (e.g., a lock release or a notify), n is said to be *ready dependent* on m [17]. In addition, ready dependences can also arise within a single thread t_m: if node p is reachable in t_m's control flow graph from m where m is either a lock acquire or wait statement, then p is ready dependent on m since m's failure to complete could cause p to never be executed.

Effects of references/aliasing: The calculation of above data dependences, interference dependences, and ready dependences both in sequential and concurrent setting, becomes much more challenging (and the results much less precise) when aliasing is introduced. In the presence of aliasing, two variables can refer to the same data/object, and so, it is possible that an update via `a.f` will affect the value of `b.f` when `a` and `b` are aliases. Our previous work [32], describes the various forms of alias and escape analyses that are used in Indus to address this issue.

3.2 Indus

The goal of the Indus project is to provide a Java library of program analyses and transformations for Java to enable sophisticated program analyses such as program slicing and program specialization via partial evaluation. In its current state, the project provides a large collection of program analyses, a Java program slicing framework, and a sophisticated user interface for the slicer [23] built as an Eclipse [13] plug-in.

Static analysis support provided by Indus includes a general flow analysis framework in which an *object-sensitive object-flow analysis* [30] is implemented that provides points-to information for objects, a *call graph analysis* of varying levels of precision, a *thread graph analysis* that calculates the method-to-thread containment relation in a given program, an *alias-aware interprocedural use-def analysis* that provides use-def information for reference type variables across procedural boundaries, and a *side-effect analysis*. Indus also provides two basic concurrency specific analyses: an *escape analysis* [32] that detects if an object allocation is thread local and a *monitor analysis* that calculates containment relation between various statements and locks in the program. Building on these analyses, a rich set of dependence analyses calculate data and control dependences in *intra-procedural, inter-procedural, intra-thread, inter-thread, non-termination sensitive*, and/or *non-termination insensitive* settings. These analyses subsume all dependences mentioned in Section 3.1. Further, the precision of interference and ready dependence analyses is improved by leveraging escape analysis to prune dependence based on non-thread local access to objects [32] and safe-lock analysis to prune divergence based dependence due to synchronization constructs [17].

4 Slicing for Model Reduction

4.1 Issues

Property-directed slicing: In earlier work [18], we presented the foundations of slicing as a model reduction technique with emphasis on the notion of "property-directed slicing" – program features mentioned in the property to be checked formed the slicing criteria, hence, leading to the automatic removal of program features that can be statically shown to be irrelevant to the property. In this paper, we evaluate the effectiveness of slicing with respect to two classes of properties: deadlock checking and assertions.

For deadlock checking, the slicing criteria consists of all synchronized and wait/notify statements. The sophisticated analyses of Indus allow us to optimize this in several ways, e.g., if Indus escape analysis indicates that objects being used in a synchronized statement are non-escaping, such statements can be omitted from the criteria since there is no contention for these locks. For assertion checking, the slicing criteria simply consists of the assertion statements in the program.

Call-graph reachability vs. slicing: Researchers familiar with static analysis might initially conceive of obtaining functionality related to slicing by simply constructing the call graph of the program to be checked and then eliminating code for methods that are not reachable in the call graph. This is especially relevant in model checking frameworks like Bandera that *translate* a program representation to a lower-level language (e.g., BIR) for model checking. Removing the code for unreachable methods via call-graph construction/analysis reduces overhead in the translation phase (code for unreachable methods does not have to be translated). Precise call graph construction in a concurrent OO language is non-trivial; it must be intertwined with points-to/alias analysis to resolve virtual method invocations, it must take control flow due to exceptions into account, and it must account for the invocation of class initializers done implicitly by the virtual machine and not by explicit invocation sites in the program.

The major drawback of mere call-graph reachability is that it does not eliminate parts of the system that are reachable in the system but do not affect the property being verified. For example, while checking for deadlocks, every reachable call to System.out.println() in the system will be included when these calls (almost always) do not affect the deadlocking behavior of the system. Inclusion of such calls increases model checking cost due to simulation of unnecessary transitions and storage of data entities required by such transitions in the state vector. On the other hand, slicing can detect and exclude such unnecessary calls and provide increased reduction in terms of model checking cost in comparison with call-graph reachability.

Bandera implements pruning of unreachable methods and data using the static analyses of Indus. Since such analyses are well-known in the static analysis community, we consider an approach that includes them as the *baseline* for our experiments: we seek to demonstrate the benefits of slicing *beyond* the already substantial reductions derived from such analyses.

4.2 Research Questions

We have focused our empirical study on four specific questions related to the use of slicing as a reduction technique.

(RQ 1) How does the net effect of applying program slicing to reduce the cost of model checking multi-threaded Java programs vary with program size and complexity?

(RQ 2) What is the incremental benefit of program slicing compared to call-graph based reachability optimizations in reducing the cost of model checking multi-threaded Java programs?

(RQ 3) What is the incremental benefit of program slicing compared to state-of-the-art partial order and thread symmetry reductions in reducing the cost of model checking multi-threaded Java programs?

(RQ 4) Does program slicing yield greater reductions for model checking assertion-based specifications of non-trivial multi-threaded Java programs in comparison to checking for deadlock?

The lack of significant experimental has left the research community with intuitions and opinions, but no definitive evidence on the effectiveness of slicing for model reduction. While some initial case studies point to the effectiveness of slicing [6, 8, 26] some experts have developed what can only be described as negative intuitions about the effectiveness of slicing as a model reduction technique. In the latter case, the best justified of these intuitions have been arrived at via formative studies of slicers implemented in verification frameworks and applied to collections of transition system descriptions as opposed to program source code. For example, Holzmann indicates that he has found SPIN's slicing capabilities useful for finding small redundancies in Promela models, but not that effective for reduction in realistic Promela properties/systems [20]. Holzmann notes that experience with program slicing on Java source code might be different, and in fact, we hypothesize that one reason why slicing has not been widely recognized as a reduction technique is that it has not been applied to realistic source code. With **(RQ 1)** we seek to provide evidence to indicate whether software model checking reductions can be achieved in analyzing realistic Java code bases. To answer this question, we have selected both relatively small, and well-studied, examples from the software model checking literature and larger examples that have not previously been analyzed via model checking.

Section 4.1 explains how sophisticated, but well-understood static analyses, could remove *unreachable/non-accessed* program components. However, these analyses do not eliminate code fragments that are reachable but yet *irrelevant* with respect to the property (e.g. writes to standard output via `System.out.println()`). Slicing is more costly and much more difficult to implement, yet it is more precise and it is able to eliminate reachable but irrelevant features. Answering **(RQ 2)** will provide evidence on both the relative cost of these two analyses and their relative benefit. Given the exponential complexity of model checking, if static call graph and slicing provide clear benefits in terms of reduction then it is likely that those benefits would outweigh any increase in static analysis cost. Such an answer would indicate that existing Java model checking should always employ call-graph analysis or program slicing as a pre-phase.

Research on reduction techniques for model checking has produced significant results over the past decades. Partial order reductions [36, 12] and heap symmetry/canonicalization reductions [21, 34] are completely automated reductions that have been applied to reduce the state space of concurrent OO software models. We believe that the model checking research community is in broad agreement that these techniques should *always* be used when model checking non-trivial programs. Note that while counter-example guided predicate abstraction [2] has been applied with good success in sequential settings where programs do not include a lot of heap data manipulation, it has yet to be applied in the concurrent OO setting in a completely automatic way. Moreover, tool-based data abstraction techniques for Java [11] still require some user intervention. Thus, in **(RQ 3)**, we seek evidence on the incremental improvement in reduction that can be achieved by slicing (a completely automated technique) relative to existing completed automated reduction techniques concurrent OO programs. We believe that slicing should only be considered worthwhile for incorporating into tools if it can provide reductions over-and-above what can be achieved by existing automated state-space reductions.

As noted in Section 4.1, slicing for model reduction is *property driven*. Answering **(RQ 4)** will provide evidence on the degree to which model reduction via program slicing is sensitive to the class of property under analysis. We have selected two classes of properties: deadlocks and assertions. A deadlocks can be considered a relatively *global* property since it is potentially related to every blocking statement in the program. It has been observed, e.g., [1, 7], that in many cases *data* manipulation can be relatively cleanly separated from *control* structures involved in synchronization. Thus, one might anticipate that significant data portions of the program could be removed when slicing for deadlock. Unfortunately, distinctions between data and synchronization are often blurred in the context of object-oriented programming where synchronization is achieved by locking data structures. Thus, slicing multi-threaded Java programs for deadlock preservation may not yield as significant a reduction as one might expect. Assertions, in contrast, are often considered *local* properties since they reference a single control point and an expression over variables in a program scope. In practice, the locality of an assertion can vary widely depending on complexity of the asserted expression. Similar expressions form the definition of *observable propositions* [9] that are used in defining temporal logic properties of programs. Thus, we expect that reduction results for assertional specifications will be indicative of the results for model checking temporal logic formulae. In answering this question, we seek to determine whether slicing on localized properties can yield additional reduction in model checking. To emphasize the *locality* of assertions we defined a collection of simple parameter and object field assertions that are enforced on method entry and that reflect comments in the source code; we did not apply slicing to assertions attempting to capture data structure properties of programs.

5 Description of Code Bases

We evaluate the effectiveness of slicing as a model reduction technique relative to the research questions stated in Section 4.2 over a collection of multi-threaded Java programs. In this section, we describe these Java code bases and characterize them in terms of static measures of their control, data and synchronization complexity. Table 1 presents the counts for several measures of each example: bytecodes, classes, methods, fields, `new` expressions, calls to `Thread.start()`, catch blocks, synchronized statements and methods, `wait` statements, and `notify`/`notifyall` statements. For each example, the first table row gives these measures both for the total of the application and library code that comprise the program and, independently, for just the application code; the values are given separated by a /. We provide the application measures for reference, but the tools process both the application and referenced library code. The second row gives the measures for the call-graph reachability based pruning from Section 4.1 and deadlock-preserving slicing (separated by '/') when applied to the complete code base (application+libraries). For example, 34479 bytecodes in the complete RAX system are reduced to 768 bytecodes by call-graph reachability and to 101 bytecodes by slicing. It is important to understand that these are static measures that the code base complexity presented to the program slicer, rather than the state space complexity presented to the model checker. For example, the static number of `Thread.start()` calls and `new` expressions typically underestimate the number of running threads and object instances in a system since these calls often appear in loops.

The set of programs includes five *standard* examples, that are commonly used in experiments in the concurrency literature, but which are best thought of as implementations of algorithm sketches rather than realistic Java code bases. *BBuffer* implements a thread-safe queue using an array and wait/notify synchronization and includes very simple producer and consumer threads. *Pipeline* implements a dynamically assembled sequence of threads that are used to parallelize a simple staged computation on integer data. *SBarbers* is a classic synchronization problem that is presented in many operating systems textbooks (e.g., [1]). *RW* includes an abstract class that implements a concurrent-read and exclusive-write policy for controlling access to program regions of implemented as methods that override abstract methods; a simple set of reader and writer threads complete this example. *RAX* is the distillation of a bug in the NASA remote experiment platform that was presented in [5].

Two *small* programs implement simple discrete-event simulators for a *DiskSched*uler and a non-trivial *AlarmClock*. In general, these examples are significantly more heap-data intensive than the standard examples.

Of the three *larger* applications we considered, one is from the literature *RepWorker* [3], one from the JavaGrande benchmark [22], *RayTracer*, and one is an Internet-scale publish-subscribe infra-structure, named *Siena*, that has been the subject of several studies of software testing techniques [10]. The *RepWorker* implements a highly-configurable client-server data-distribution framework with a sample Jacobi relaxation example running atop it as an application. The JavaGrande benchmark implements a non-trivial scientific calculations whose synchronization is achieved through the use of barriers.

Table 1. Static measures of examples

Name	Bytecode	Class	Method	Field	New	Thrd	Except	Synch	Wait	Nfy
AlarmClock	34728/319	349/6	3245/25	1295/19	1461/13	4/3	473/18	252/15	3/2	3/2
Reach/Slice	1020/345	49/25	98/41	57/18	45/18	3/3	23/17	18/15	2/2	2/2
BBuffer	34577/168	351/8	3238/18	1294/8	1455/7	4/3	462/7	241/4	3/2	3/2
Reach/Slice	873/194	50/23	91/30	151/15	38/10	3/3	12/7	8/5	2/2	2/2
DiskSched	35181/793	348/5	3238/18	1297/21	1510/63	2/1	458/3	239/2	3/2	2/1
Reach/Slice	1314/643	48/25	92/36	52/15	79/52	1/1	12/7	8/5	2/2	1/1
Pipeline	34475/66	347/4	3229/9	1280/4	1452/4	3/2	456/1	239/2	2/1	2/1
Reach/Slice	764/97	46/19	80/20	42/6	35/7	2/2	6/1	6/3	1/1	1/1
SBarber	34564/155	347/4	3229/9	1287/11	1455/7	3/2	474/19	245/8	4/3	4/3
Reach/Slice	853/184	46/19	80/20	48/12	38/10	2/2	24/19	12/9	3/3	3/3
RAX	34479/70	347/4	3229/9	1283/7	1452/4	3/2	456/1	239/2	2/1	2/1
Reach/Slice	768/101	46/19	80/20	45/9	35/7	2/2	6/1	6/3	1/1	1/1
RW	34699/290	348/5	3246/26	1287/11	1455/7	5/4	473/18	247/10	3/2	3/2
Reach/Slice	945/268	47/20	95/34	49/13	39/11	4/4	23/18	14/11	2/2	2/2
RepWorker	34885/574	356/14	3265/47	1317/42	1471/23	2/1	460/5	255/18	6/5	7/6
Reach/Slice	1792/1460	69/57	158/131	82/47	81/74	1/1	22/19	6/6	5/5	6/6
RayTracer	35544/1783	361/19	3327/109	1351/76	1538/91	2/1	486/31	142/15	1/1	1/1
Reach/Slice	2757/2520	72/65	195/179	126/95	119/114	1/1	29/26	12/12	1/1	1/1
Siena	49314/9229	489/74	4700/620	1688/296	2119/389	5/4	644/186	384/139	10/7	14/8
Reach/Slice	14213/13491	198/194	844/909	424/306	574/565	3/3	164/164	105/110	4/4	8/7

As the data indicate, slicing can yield reductions ranging up to two orders of magnitude in the size of the code base to be analyzed by the model checker. While the call-graph reachability reduction can also yield significant reductions, program slicing always achieves better reductions. As we will see in the next section, the elimination of additional statements and fields can give rise to substantial reductions in model check times and more than compensate for the relatively modest slicer run-time which for even the largest examples was at most several seconds.

6 Experimental Results

In this section, we report on the performance of model checking selected properties of the examples described in Section 5 using different state-space reduction options. We present statistics from different subsets of these model checking runs and discuss how those results help to answer the research questions from Section 4.2.

Table 2 presents a sampling of the model check runs we performed. For the systems we studied, we model checked a total of 31 different variations of those systems; variations were either applying fixes to known bugs or activating additional specifications encoded as assertions in the code base. To conserve space, we only list runs for variations on a given program that are substantially different in performance. We differentiate runs that found errors from those that did not by using an *e* subscript on example names; note that several of our examples have buggy versions. For each system variation we

Table 2. Effect of slicing on model checking

Name	Conf.	States	Trans.	Memory	Time
AlarmClock$_e$	dR	11204	28931	2.45	25.2
	dRP	106	870	0.12	11.1
	dS	4867	11593	0.12	13.1
	dSP	83	693	0.12	10.3
AlarmClock	dR	1469917	5117602	4.49	57:56.4
	dRP	2305	49801	2.55	50.6
	dS	724666	2501705	3.69	13:1.6
	dSP	1204	25298	2.19	20.1
	aS	724666	2501705	3.22	12:56.4
	aSP	1204	25298	2.13	20.4
BBuffer	dR	36405	90882	3.9	1:1.2
	dRP	138	3980	0.12	14.4
	dS	7484	18578	0.11	13.9
	dSP	28	630	0.12	10.7
	aS	7484	18578	0.12	13.9
	aSP	28	630	0.12	10.2
DiskSched$_e$	dR	7687219	2044564	5.71	9:48:1.8
	dRP	7690	858963	4.16	30:59.7
	dS	5487745	14302033	5.88	1:16:34.5
	dSP	7688	816991	1.81	5:17.6
	aS	5487745	14302033	5.88	1:16:26.7
	aSP	7688	816991	3.65	5:11.8
Pipeline	dR	9892140	43821449	5.52	5:41:5.7
	dRP	7379	76307	4.21	45.6
	dS	9881030	43771450	5.43	5:28:4.4
	dSP	7379	76303	4.21	40.6
SBarber$_e$	dR	197	197	0.12	13.4
	dRP	31	197	0.12	11.6
	dS	193	193	0.12	10.7
	dSP	31	193	0.12	12.1
RAX$_e$	dR	268	279	0.80	28.4
	dRP	33	252	0.80	27.2
	dS	266	277	0.57	26.8
	dSP	33	250	0.84	26.2
RW	dR	25197913	26780595	5.86	13:21:24.6
	dRP	603	44621	4.13	1:11.2
	dS	113727	415098	4.15	1:29.8
	dSP	134	2103	0.12	11.3
RepWorker	dR	2818232	8102858	4.77	2:9:59.9
	dRP	3091	96798	1.65	2:43.2
	dS	2736984	7867615	4.74	1:25:34.9
	dSP	2676	90735	3.52	2:1.1
RayTrace	dR	2404257*	4754057*	6.26*	20:0:0*
	dRP	11610	2923881	5.35	9:8:45.4
	dS	2803535*	5552933*	6.34*	20:0:0*
	dSP	10932	2776238	5.18	6:53:13.9
Siena$_e$	dR	11465	11475	2.8	9:37.4
	dRP	116	11475	3.86	12:1.2
	dS	11310	11319	3.93	6:30.4
	dSP	114	11319	4.13	7:51.5

ran the model checker in at least four configurations. In each of these configurations, the model checker was configured to terminate when the first error was encountered in the state space search. The tables list statistics for different configurations of the model checking tools where *d* indicates deadlock check, *a* indicates assertion check, *R* indicates call-graph reduction, *S* indicates slicing, and *P* indicates the use of POR. For each configuration, we give the total number of stored states, the number of transitions searched, the maximum memory consumption of the toolset in giga-bytes, and the total run-time for the toolset in hours:minutes:seconds format. The model checks were run as the only application on an Opteron 250 processor with 12 Gigabytes of RAM running Linux using the SDK 1.5. The small number of runs that exceeded 20 hours were terminated and they are noted in the table with *. Consequently, the reduction results presented below should be viewed as lower bounds.

6.1 Analysis of Data and Research Questions

Our analysis of the experimental data confirms that slicing is a cost-effective state space reduction technique, however the data do illustrate some of the limitations of program slicing and suggest opportunities for additional approaches to refining slicing to achieve greater reductions.

To assess the cost-effectiveness of slicing we calculate *reduction factors* that capture the ratio of the total run-time of our toolset for specific pairs of configurations. The mean of a reduction factor is calculated over a specified subset of the 31 model check runs and positive outliers are removed; a positive outlier is a reduction factor that exceeds the mean by more than two standard deviations. We consider different pairs of configurations of the model checker to address the different research questions. In the discussions below, we focus on run-time reduction factors as opposed to factors related to memory consumption or state-space size, since this measure best captures the total time to apply slicing reductions which happen before the actual model checks run. We have observed, however, that space reductions seem to follow the same trend as time reductions.

(RQ 1) is concerned with the variation in effectiveness of slicing as a reduction with program size and complexity. For this question, we consider the reduction factor dR/dS which is a measure of the effectiveness of slicing. To assess this question, we group the examples into the set of *larger* examples and the rest. The mean of dR/dS over set of larger examples is 1.4 and for the smaller examples it is 2.7.

This data seems to suggest that slicing scales poorly as a state-space reduction. It is well-understood, however, that static code measures, such as lines of code, are poor indicators of state-space size and our real intention is to understand the reduction of slicing as the size of the state-space scales. Furthermore, we believe that the real benefit of slicing, and of many state-space reductions, is only apparent when the search is *stateful*. To assess this, we compared the number of matched states to the number of stored states during model checking. We term a model check run *sparse* if the ratio of matched to stored states is less than 0.1. Most, but not all, of our error runs were classified as sparse, for example, there were runs of SBarber, RAX and Siena that required no or only minimal backtracking to find the error. The non-sparse searches included all of the model checks that verified the property being checked and several error revealing checks where the error was found late in the search. Recalculating the mean of dR/dS for this grouping of runs yields reduction factors of 1.9 for sparse and 4.0 for non-sparse searches. Even without this secondary analysis, it is clear that program slicing is an effective reduction technique since the reduction factors account for the cost of slicing.

(RQ 2) is concerned with the relative effectiveness of call-graph reachability and slicing model reductions. The mean of dR/dS across all runs in our study was 2.5 indicating a non-trivial net benefit to slicing.

(RQ 3) is concerned with the relative effectiveness of partial-order reduction and slicing. The mean of dRP/dSP over the total set of runs in our study was 2.1; the means for sparse and non-sparse subsets of runs were 1.2 and 2.9, respectively. Clearly slicing yields non-trivial additional reduction over POR. POR appears to be a more *powerful* than slicing based on the fact that the mean of dR/dRP over the total set of runs in our study was 48.3; the means for sparse and non-sparse subsets of runs were 1.2 and 105.5, respectively. Not surprisingly, POR and slicing appear to both provide benefit only when a substantial portion of the state space is searched.

(RQ 4) is concerned with the relative reduction power of slicing when the property being analyzed is deadlock or a simple assertion. The data indicate that there is no statistically significant difference. For the set of examples on which both deadlock and assertional specifications were checked the mean of dRP/dSP was 2.8 and the mean of aRP/aSP was 2.7. This was surprising to us since we expected that a more localized specification would require less of the program to be included in the slice. In the programs we studied, however, there always existed a chain of dependences from the assertion expression to a program point that is part of the programs *synchronization skeleton*. Once one such point is drawn into the slice so to is the rest of the skeleton. This is an unfortunate consequence of high-degree of synchronization coupling in the set of multi-threaded Java programs we studied.

7 Related Work

Since its development, the concept of slicing has been applied to a wide variety of problems including: program understanding, debugging, differencing, integration, and testing; we refer the reader to Tip's survey article for a broad view of slicing [38]. Here, we focus on other work related to slicing for model reduction and verification and slicing Java programs.

Millett and Teitelbaum [26] study static slicing of Promela (the model description language for the model-checker SPIN [19]) and its application to model checking, simulation, and protocol understanding. Their work formed the basis of the Promela slicing framework that is now included in the SPIN distribution. Both [26] and the SPIN Promela slicer support slicing with criteria formed from assertions and never claims (and thus LTL formulae), but do not include support for slicing to preserve deadlock. Slicing systems in a modeling language like Promela is considerably simpler than slicing Java programs due to the absence of challenging features like heap-allocated data, exceptions, methods and dynamic dispatch, threads associated with object references, etc. However, care must be taken to deal correctly with Promela features such as channels, richer intra-thread non-deterministic choice constructs, and more sophisticated notions of blocking due to Promela's rich guarded command language (as we noted earlier, Java programs can only block due to lock acquisitions and `wait` statements.

The IF Validation Framework [4] also provides a slicing capability for the IF modeling language which is similar to Promela in its level of abstraction. In a case-study of using IF to verify properties of the MASCARA protocol for a wireless asynchronous transfer protocol, Graf and Jia [24] report reductions from 1-2 orders of magnitude for four different properties of the protocol while acknowledging that it is difficult to make general conclusions about the effectiveness of slicing since amount of reduction depends significantly on the particular property and system considered.

Clarke *et al.*[6] present a tool for slicing VHDL programs with dependence graphs. Using the VHDL description of the controller logic for a RISC processor and two accompanying CTL properties (a safety and a liveness property), they show that slicing reduces the reachable state space from roughly 10^{38} states to 10^{22} states. They also observe a *reverse scalability* effect for slicing – smaller VHDL programs tend to have fewer irrelevant components, and thus the benefits of slicing to improve (percentage-wise) as programs grow in size. Sen *et al.*[35] use a substantively different technique called *computation slicing* for model reduction in verification of "systems on a chip" hardware designs.

8 Conclusion

Most researchers have developed strong opinions about the potential effectiveness of program slicing as a reduction technique for software model checking. Many of those opinions are in the negative. We believe that the study presented in this paper provides convincing evidence that slicing is efficient to apply, taking no more than 40 seconds on even the largest code base we considered, and yields non-trivial reductions in model check time, averaging a factor of 4 improvement for non-trivial model checks. Given the long-running nature of model checks, this magnitude of reduction can significantly increase productivity. Furthermore, these reductions are orthogonal to existing state-space reductions and can thus be considered an extension to the state of the art.

As with any experimental study, one can question the external validity of these conclusions, and we plan to increase the number of examples in our study and vary the sources from which we draw those examples to provide more evidential force to our findings. In spite of such questions, given that we never encountered a model check run where slicing caused a non-trivial increase in run-time in our study, we believe that concluding that slicing is a cost-effective model checking reduction is justified.

References

1. G. R. Andrews. *Concurrent Programming: Principles and Practice*. Addison-Wesley, 1991.
2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, pages 203–213, June 2001.
3. Bandera. +http://bandera.projects.cis.ksu.edu+. SAnToS Laboratory.
4. M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF: A validation environment for timed asynchronous systems. In *12th International Conference on Computer Aided Verification (CAV 2000)*, LNCS 1855, pp. 543–547, July 2000.
5. G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder – A second generation of a Java model-checker. In *Proceedings of the Workshop on Advances in Verification*, July 2000.
6. E. Clarke, M. Fujita, S. Rajan, T.Reps, S. Shankar, and T. Teitelbaum. Program slicing of hardware description languages. In *Proceedings of CHARME'99*, September 1999.
7. E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer.
8. J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, June 2000.
9. J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing checkable properties of dynamic systems: The Bandera Specification Language. *International Journal on Software Tools for Technology Transfer*, 2002.
10. H. Do, S. Elbaum, and G. Rothermel. Infrastructure support for controlled experimentation with software testing and regression testing techniques. In *2004 International Symposium on Empirical Software Engineering (ISESE 2004)*, pages 60–70. IEEE Computer Society, 2004.
11. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported program abstraction for finite-state verification. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.
12. M. B. Dwyer, J. Hatcliff, V. R. Prasad, and Robby. Exploiting object escape and locking information in partial order reductions for concurrent object-oriented programs. *Formal Methods in System Designs*, 25(2–3):199–240, September–November 2004.
13. Eclipse Consortium. Eclipse website. `http://www.eclipse.org`.

14. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 110–121, Long Beach, California, USA, January 2005. ACM.
15. P. Godefroid. *Partial Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.
16. A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21. ACM Press, 2002.
17. J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *Proceedings of the 6th International Static Analysis Symposium (SAS'99)*, volume 1694 of *Lecture Notes in Computer Science*, Sept. 1999.
18. J. Hatcliff, M. B. Dwyer, and H. Zheng. Slicing software for model construction. *Journal of Higher-order and Symbolic Computation*, 13(4):315–353, 2000.
19. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–294, May 1997.
20. G. J. Holzmann. Personal communication, Oct. 2005.
21. R. Iosif. Symmetry reduction criteria for software model checking. In *Proceedings of Ninth International SPIN Workshop*, volume 2318 of *Lecture Notes in Computer Science*, pages 22–41. Springer, Apr. 2002.
22. Java Grande Benchmarking Project. Java Grande forum benchmark suite – thread version 1.0. `http://www.epcc.ed.ac.uk/computing/research_activities/java_grande/`.
23. G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering Indus Java program slicer to Eclipse. In *Proceedings of the Fundamental Approaches to Software Engineering, FASE 2005*. Springer, April 2005.
24. G. Jia and S. Graf. Verification experiments on the MASCARA protocol. In M. B. Dwyer, editor, *Model Checking Software: 8th International SPIN Workshop*, volume 2057 of *LNCS*, pages 123–142, Toronto, Canada, May 2001. Springer.
25. J. Krinke. Static slicing of threaded programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).
26. L. I. Millett and T. Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *Proceedings of the 4th International SPIN Workshop*, LNCS, 1998.
27. M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 180–190, 2000.
28. A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(8):965–979, 1990.
29. V. P. Ranganath. Indus. +http://indus.projects.cis.ksu.edu+.
30. V. P. Ranganath. Object-flow analysis for optimizing finite-state models of Java software. Master's thesis, Kansas State University, 2002.
31. V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005*. Springer, April 2005.
32. V. P. Ranganath and J. Hatcliff. Pruning interference and ready dependences for slicing concurrent Java programs. In E. Duesterwald, editor, *Proceedings of Compiler Construction (CC'04)*, *Lecture Notes in Computer Science* 2985, pages 39–56. March 2004.

33. Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. In *Proceedings of the 9th European Software Engineering Conference / 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.
34. Robby, M. B. Dwyer, J. Hatcliff, and R. Iosif. Space-reduction strategies for model checking dynamic systems. In *Proceedings of the 2003 Workshop on Software Model Checking*, July 2003.
35. A. Sen, J. Bhadra, V. K. Garg, and J. A. Abraham. Formal verification of a system-on-chip using computation slicing. In *International Test Conference ITC*, pages 810–819, October 2004.
36. S. Stoller. Model-checking multi-threaded distributed Java programs. In *International Journal on Software Tools for Technology Transfer*. Springer, 2002.
37. D. Suwimonteerabuth, S. Schwoon, and J. Esparza. jMoped: A Java bytecode checker based on Moped. In *Proceedings of the 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, number 3440 in Lecture Notes in Computer Science, pages 541–545, 2005.
38. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
39. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – A Java optimization framework. In *Proceedings of CASCON'99*, Nov. 1999.