Pert: The Application-aware Tailoring of Java Object Persistence

Peng Liu, Charles Zhang *Member, IEEE*

***Abstract*—Persistence is a widely used technique which allows the objects that represent the results of lengthy computations to outlive the process that creates it, in order to considerably speed up subsequent program executions. We observe that conventional persistence techniques usually do not consider the application contexts of the persistence operations, where not all of the object states are necessary to be persisted. Leveraging this observation, we have designed and implemented a framework called Pert, which first performs static program analysis to estimate the actual usage of the persisted object, given the context of its usage in the program. The Pert runtime uses the statically computed information to make efficiently tailoring decisions to prune the redundant and unused object states during the persistence operations. Our evaluation result shows that the Pert-based optimization can speedup the conventional persistence operations by 1 to 45 times. The amount of persisted data is also dramatically reduced, as the result of the application-aware tailoring.**

***Index Terms*—Object persistence, program analysis, performance optimization.**

I. Introduction

Object persistence is a technique that stores the runtime states of objects to the non-volatile storage such as file systems or database systems. Through the object persistence, the runtime object states, often representing results of lengthy and expensive computations, can live across different computation processes. Object persistence is widely and actively used. It is a required functionality for the Enterprise JavaBeans (EJB) architecture and used in RPC services to provide the backup, restore, and replication services [1]. Object persistence is also one of the fundamental techniques used in the program checkpointing [19] to restart the execution of programs at specified points. The performance of object persistence frameworks has a deep impact on the performance of many critical software systems.

Object persistence can be achieved by two general approaches. We can either dump the contents of the physical memory pages at the levels of OS or VM [14], or we can implement object persistence as the user application behavior. The first approach is tightly coupled with the features of the underlying platform. The second approach is more portable and flexible, hence, widely implemented in general purpose frameworks including the JRE object output stream (OOS) library, the XStream library, the Hibernate library, and many others [2]. Given a target variable, those frameworks usually traverse the entire referenced object graph and persist all of the related fields.

As commonly observed [6], [19], performing object persistence at the application level can be an expensive operation. For instance, the JRE OOS library traverses the entire object graph under the instruction of the class meta-data[3]. Therefore, numerous optimizations are proposed to reduce the traversal cost. Philippsen et al. [13], as well as the protocol buffer approach, uses compiler to generate the explicit persistence code instead of using the reflection-based class meta-data. The compact encoding of objects, such as the approach of JSON [4], can also be used to speed up the persistence operation. However, these optimizations, designed for general persistence purposes, are not aware of specific application usage scenarios, where only a subset of the object fields need to be persisted. The state of art optimization [4] by Elbaum et al. is aware of the application-specific usage scenarios by carrying out an intra-object analysis to optimize the persistence for the unit testing. However, their usage scenarios, which are limited to the scope of the current function of interest, are not general. Consequently, they miss other application-aware optimization opportunities when the whole program is being considered.

In this paper, we present a new optimization algorithm based on the observation that the amount of the persistence work can be tailored to its specific usage contexts. Specifically, we observe two general persistence scenarios that give rise to good optimization opportunities. These scenarios are exemplified in Figure 1 using the code fragments [5] of the OO7 benchmark [3], a standard benchmark for object-oriented databases with persistence calls inserted. The first scenario illustrates that, when saving a previously persisted object, we only need to persist what has been incrementally modified. As demonstrated in Figure 1, the `updateAssemblyParts` operation persists the `BaseAssembly` object before performing a long update operation (line 3-line 7). It then persists the object again (line 8). Since the operation only modifies the collection object (`assm.unsharedParts`) by adding `Composite` objects, we could only persist the changed part (`assm.unsharedParts`) at the second persistence operation. The second scenario tries to identify object states that are not further used after a persistence operation. In Figure 1, the method `query5` loads `BaseAssembly` objects from the disk and queries the `buildDate` field. Since no other fields of the `BaseAssembly` object are used, we can shorten the costly reloading process and, symmetrically, the persistence operation, by ignoring the unused fields. Please note that the examples are deliberately simplified for the purpose of illustration. In practice, persistence operations can be placed at arbitrary places in the code, and all possible program execution paths need to be considered.

These scenarios have many real-world applications. The first scenario commonly exists in the application which requires multiple persistence of the same objects. In reliable execution [7],

[1]EJB usage. URL:http://www.oracle.com/technetwork/java/javaee/ejb/index.html RPC services. http://www.corba.org/ http://java.sun.com/products/jms/

[2]The Java object output stream utility. URL:http://java.sun.com/j2se/1.4.2/docs/api/java/io/ObjectOutputStream.html The XStream Library. URL:http://xstream.codehaus.org/ The Hibernate library. URL:http://www.hibernate.org/

[3]The class meta-data is essentially a descriptor of the layer out of the class.

[4]JSON URL:http://www.json.org/

[5]For the ease of presentation, we used abbreviated class names and compact representations of control flows in the original code.

```
private void updateAssemblyParts(BaseAssembly assm){
        persist(assm);
        for (int i = 0; i < numComponents(); i++) {
                Long partId = getSeqId();
                Composite composite = new Composite(partId);
                assm.addUnshared(composite);
        }
        persist(assm);
}

public void query5() {
        ...
        int num = 0;
        Query q = sess.createQuery("select ba
                  from BaseAssembly ba ");
        Iterator qIter = q.list().iterator();//the reloading code is inside
        while (qIter.hasNext()) {
                BaseAssembly ba = qIter.next();
                ba.getBuildDate();
                num++;
        }
        ...
}
```

Fig. 1. Object persistence examples. The `persist` operations are inserted.

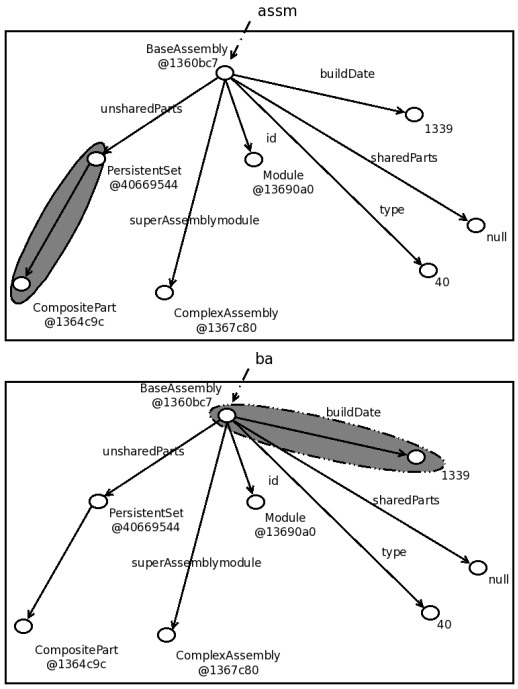


Fig. 2. Heap Layout (The shaded region represents the actual states required to be persisted)

one can *regularly* persist the runtime program states so that they can be safely recovered once the execution crashes. For efficiency, the object states which are not incrementally modified can be ignored by the persistence process. In component-based software testing [15] [4], one needs to persist the states at the component boundaries and reload the states to test each component individually. As a result, the objects shared by the components are persisted multiple times. Such operations can be more efficient if we only persist the modified object states. The second scenario is applicable to techniques such as the delta debugging [20]. Delta debugging on large programs is a costly computation. Instead, one can partition the program into several partitions and apply the delta debugging on each partition [19]. As each partition differs in terms of the objects of debugging interest, some declared fields may not be used in a partition and hence can be ignored when the objects are persisted and reloaded.

Leveraging these observations, we have designed and implemented Pert [6], an object persistence framework that generates tailored persistence operations using the context information of where the persistence operations are performed. Based on the two general scenarios illustrated earlier, we present two static optimization algorithms, the *Incremental Modification (IM)* analysis and the *Use-Directed Reduction (UDR)* analysis, which make persistence operations adaptive to their application contexts. The *IM* analysis locates the updated object states which should be persisted again, through analyzing the incremental modification of object states in the program. The UDR analysis thoroughly examines the possible future execution paths with respect to a particular persistence operation and determines the parts of the heap that are necessary to be saved. Both analysis are performed for the whole program in the context-sensitive and inter-procedural manner. Central to these analysis is the concept of the tagged abstract heap graph (TAHG), statically computed to conservatively approximate the runtime object graphs [7]. In TAHGs, the nodes represent objects referenced by the fields, and the directed edges the field references, as computed by the pointer analysis. Using the persistence operations as the demarcation points, our static analysis first examines the relevant code regions and constructs corresponding TAHGs to encode the side-effect information. The Pert runtime tracks the calling context of the persistence operations and retrieves the corresponding TAHGs to decide what object states need to be persisted.

We evaluate Pert using 13 third-party subject programs from benchmarks such as SpecJVM and SpecJBB, carefully instrument them with persistence operations around the transactional code. Using object serialization as a specific persistence mechanism, our results show that, compared to conventional persistence approaches, the use of Pert not only preserves the correctness of the benchmark computations. It also reduces the persisting and the reloading time by 1X to 45X. The reduction of the size of the persisted data ranges from 7% to 99%. In addition, we show that our optimization is independent of the specific persistence mechanisms and also applicable to the popular object-relational persistence mechanism. Evaluated using the OO7 benchmark and the Hibernate framework, we observed around 20% reduction of both the persisting and the reloading time.

We make the following research contributions to the state of the art of object persistence technology:

1) We contribute both the incremental modification analysis and the use-directed reduction analysis as two general algorithms to adaptively subset the runtime object graph for the need of the persistence operations, without compromising the computational correctness.
2) We contribute the Pert framework, consisting of a set of utilities that enable the persistence tailoring using a hybrid approach. The Pert framework combines static analysis with efficient runtime support to pay a low runtime overhead for the tailored persistence operations.
3) We evaluate the correctness and the performance of the Pert-based persistence operations using standard Java bench-

[6] Pert is an acronym for "**Per**sistence **t**ailoring"

[7] Figure 4(b) shows examples of the runtime object graph

marks. We compare to the conventional persistence approaches and quantify the performance speedup of persistence operations and the size reduction of the data to be persisted.

II. Application-aware Persistence Tailoring

For an object to be persisted, Pert aims at generating the tailored persistence code, which saves the object state information that is "just enough" for the correct subsequent computation of the program. We achieve this goal by first performing the static side-effect analysis on the heap states of the persisted object. The analysis results are encoded in a special data structure, the *tagged abstract heap graph (TAHG)*, used by the Pert runtime to make adaptive and efficient persistence decisions. In the following sections, we first formally present the definitions and the models used in our algorithm. We then present the details of two key persistence analysis: the incremental modification analysis and the use-directed reduction analysis. We also discuss the design of the Pert runtime persistence support.

A. Definitions

We first formally define a few key concepts to facilitate the comprehension of our technique. To better elucidate these concepts, we use an on-going example, comprised of a hypothetical calling relationship, depicted in Figure 3, that involves four methods and the persistence call `persist`.

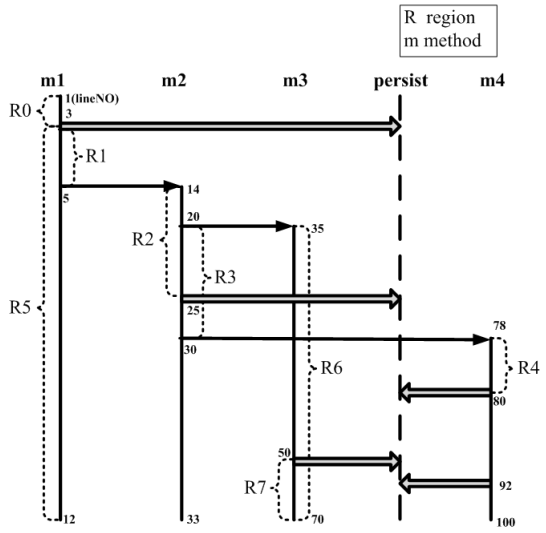


Fig. 3. The calling relationships of four methods. The vertical bars represent the code regions of the method body. The arrows signify method calls at the respective call sites annotated by the line number. The thick arrows highlight the persistence calls.

Definition 1 (Persisting call)**.** *A persisting call, $\widetilde{\xi_p^o}$, is defined as follows.*

$$\widetilde{\xi_p^o} := \xi_1 \rightsquigarrow \xi_2 \rightsquigarrow ... \rightsquigarrow \xi_n \rightsquigarrow \xi_p^o,$$

where ξ_i denotes a procedure call and ξ_p^o the call to the persisting API that saves the state of the object o.

Note that, in our definition, we define a persisting call using its calling context information: $\xi_1 \rightsquigarrow \xi_2 \rightsquigarrow ... \rightsquigarrow \xi_n$. We encode the call site, ξ_i, using a tuple $\langle methodname, linenumber \rangle$. In Figure 3, $\langle m1, 5\rangle \rightsquigarrow \langle m2, 25\rangle$ is an instance of a persisting call.

The persisting call, ξ_p^o, is often paired with the reloading call, ξ_r^o, which restores the states of an object from the non-volatile storage to the heap memory referred to by the variable o. We define such calls as $\widetilde{\xi_r^o}$ in the same fashion as $\widetilde{\xi_p^o}$. We refer to both calls as persistence calls in general.

Definition 2 (Post-Region)**.** *The post-region of $\widetilde{\xi_p^o}$, and equivalently, of $\widetilde{\xi_r^o}$, is defined as:*

$$post(\xi_p^o) := tail(\xi_1) \cup tail(\xi_2) \cup ... \cup tail(\xi_n) \cup tail(\xi_p^o),$$

where $tail(\xi_i)$ stands for all program statements reachable from ξ_i (ξ_i is not included) in the inter-procedural *control flow graph of the method where ξ_i is lexically located.*

The post-region corresponds to the static approximation of the code possibly executed after the call, $\widetilde{\xi_p^o}$. For example, the post-region of the call at $\langle m1, 3\rangle$, $post(\langle m1, 3\rangle)$, includes statements in both regions denoted by $R5$ and $R5$'s transitive callees. Note that, the regions $R2$, $R3$, $R4$, $R6$, $R7$ are contained in those statements.

Definition 3 (Pre-region)**.** *The pre-region of $\widetilde{\xi_p^o}$, and equivalently, of $\widetilde{\xi_r^o}$, is defined as:*

$$pre(\xi_p^o) := head(\xi_1) \cup head(\xi_2) \cup ... \cup head(\xi_n) \cup head(\xi_p^o),$$

where $head(\xi_i)$ stands for all program statements reaching ξ_i (ξ_i is not included) in the inter-procedural control flow graph of the method where ξ_i is lexically located.

As exemplified in Figure 3, the pre-region for the persistence call, $\langle m2, 25\rangle$, includes the code in regions $R0, R1$, $R2$ and their transitive callees.

Definition 4 (Object Graph)**.** *An object graph, $OG(o)$, is a graph G with a single root representing the object o. The vertex set, V, of the graph consists of a set of nodes and leaves where v_{node} represents the objects and v_{leaf} represents the primitive values such as $float$ or $integer$. The edge, e, represents a reference relationship between the nodes.*

Our definition of the object graph, similar to that of Fetzer [5], represents the recursive layout of the heap memory of an object with respect to its type definition in Java. Figure 4(b) is an example of the object graph $OG(list)$ rooted at the variable `list`. Given this root variable, we use a dotted notation to refer to a vertex in the object graph. For instance, the notation $list.head.content$ refers to the memory location pointed by the variable $content$. References to the memory locations encoded in this fashion form *reference chains*.

Definition 5 (Abstract Heap Graph)**.** *The abstract heap graph, denoted as $AHG(o)$, is a graph with a single root represented by the reference variable* **o***. The edges of AHG represent the field reference relationship between objects. The nodes of AHGs are* **point-to** *sets, computed by the static pointer analysis, representing the possible allocation sites of the objects referenced by the fields. The leaves represent primitive values. We refer to the node of the AHG as a* **cell***.*

The AHG, similar to the *abstract heap model* of Marron [9],

is a static representation of the runtime intra-object reference hierarchy rooted at an object reference. The main difference, as compared to the object graph, is that the static pointer analysis can not distinguish different runtime objects that are created at the same allocation site. Therefore, in the AHG, all objects created at the same site are represented by one allocation node (*anode*) in the point-to set. This difference between an OG and an AHG can be illustrated by Figure 4, which shows both the object graph (Figure 4:b) and the corresponding AHG (Figure 4:c) where the objects reached through the reference chain, $list.head.next$, $list.head.next.next$, are collapsed to one *anode*, hence, in the same cell of the AHG.

B. Persistence Tailoring with Tagged AHGs

The tailoring capability of Pert is based on two static analysis techniques: the incremental modification analysis and the use-directed reduction analysis. These two algorithms can be independently applied without affecting each other. We now discuss these two techniques in detail.

1) Incremental Modification: The incremental modification (IM) analysis aims at only persisting the object states modified between two persisting calls. To achieve efficiency, we approximate statically the incremental modification information to dramatically reduce the amount of runtime traversal of the OG. As illustrated in Figure 5, the IM analysis consists of three steps: the delta region analysis, the side-effect analysis, and the tagging of the AHGs.

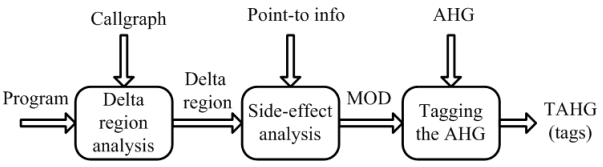


Fig. 5. Analysis phases for incremental persistence

Delta region analysis The goal of the delta region analysis is to statically identify the program statements that are possibly executed between a given pair of persisting calls. And we collectively refer to these statements as a delta region, which we formally define as follows.

Definition 6 (Delta Region)**.** *Given two variables a and b, which are aliases to the same heap object, and two persisting calls, $\widetilde{\xi_p^a}$ and $\widetilde{\xi_p^b}$, that share the same entry point, i.e., the initial calling context: $\xi_1 \rightsquigarrow \xi_2 \rightsquigarrow ... \rightsquigarrow \xi_i$, the delta region, $delta(\widetilde{\xi_p^a}, \widetilde{\xi_p^b})$, is defined as follows* [8].

$$delta(\widetilde{\xi_p^a}, \widetilde{\xi_p^b}) := between(\xi_{i+1}^a, \xi_{i+1}^b) \cup post(\xi_{i+2}^a \rightsquigarrow \cdots \rightsquigarrow \xi_n^a \rightsquigarrow \xi_p^a) \cup pre(\xi_{i+2}^b \rightsquigarrow \cdots \rightsquigarrow \xi_m^b \rightsquigarrow \xi_p^b)$$

where $between(\xi_{i+1}^a, \xi_{i+1}^b)$ stands for the statements which can be reached from the call site ξ_{i+1}^a to the call site ξ_{i+1}^b (ξ_{i+1}^a and ξ_{i+1}^b are not included) in the inter-procedural CFG of the method where both ξ_{i+1}^a and ξ_{i+1}^b are hosted.

An example of delta region in Figure 3 is the one demarcated by two call chains $\langle m2, 20\rangle\langle m3, 50\rangle$ and $\langle m2, 30\rangle\langle m4, 80\rangle$. The delta region is then defined as $between(\langle m2, 20\rangle, \langle m2, 30\rangle) \cup post(\langle m3, 50\rangle) \cup pre(\langle m4, 80\rangle)$ and includes regions $R3 \cup R4 \cup R7$.

[8]Here we slightly abuse the notation. We use ξ_i^a to represent a (non-persistence) call in the call chain associated with $\widetilde{\xi_p^a}$ in order to differentiate from calls in $\widetilde{\xi_p^b}$.

procedure: ModDelta
input: δ : `deltaregion`

```
1: for statement s ∈ δ do
2:   defValues=s.getDefValues()
3:   for value v ∈ defValues do
4:     register_side_effect(v)
5:   end for
6: end for
```

procedure: register_side_effect
input: `DefValue v`

```
1: if v is an instance field, i.e. o.f then
2:   anodes = getAnodes(o)
3:   for anode α ∈ anodes do
4:     anodeToFields.lookup(α).add(f)
5:   end for
6: else if v is an array element, i.e. a[i] then
7:   anodes = getAnodes(a)
8:   for anode α ∈ anodes do
9:     anodeToFields.lookup(α).add(FakedField)
10:  end for
11: else if  v is a static field, i.e. class.sf then
12:   classToFields.get(class).add(sf)
13: end if
```

Fig. 6. calculation of side effect in deltaregion

From the definition, our analysis is restricted to persisting call pairs that persist the same object on the heap and share the same entry point designated by the call chain: $\xi_1 \rightsquigarrow \ldots \xi_i$. As expected, ξ_1 is the first call site of the call chain in the `main` function. The output of this step is all persisting call pairs and their associated delta regions, which are used as the inputs of the following phases.

Side-effect analysis in the delta region Given a particular delta region, we traverse its program statements and examine the field store operations. For each of the modified field, e.g., `o.f`, we first retrieve the anodes that represent the base variable `o` and record the field `f` of these anodes. This process is implemented in the procedure `ModDelta`, shown in Figure 6, where `anodeToFields` is a map of modified fields indexed by anodes. `classToModList` is the other map associating the modified static fields to their corresponding class type definitions. The `ModDelta` procedure traverses the statements in the delta region, and invokes the `register_side_effect` procedure (line 4) to map the field modification to the anodes of the base pointer. The reason for registering the modification information on the anode rather than on the pointer variable is to avoid the inconsistency caused by aliases.

Tagging the AHG As afore-explained, the AHG is a static summary of the transient object graphs during the program execution. Tagged with the modification information, the AHG can be used to instruct the traversal of the runtime persistence process. In particular, when the persistence process is to visit a field reference in the OG, it first inspects the corresponding edge in the TAHG, then depending on its type, the process carries out proper persistence actions towards the OG.

The tagging process proceeds as follows. Given a delta region

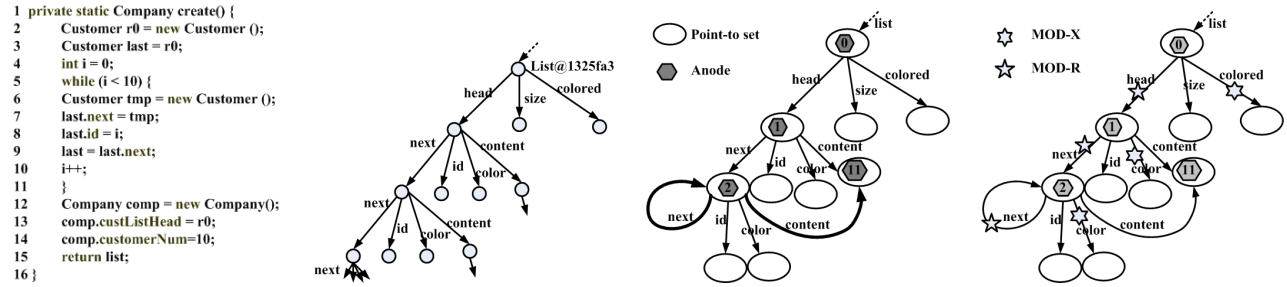


Fig. 4. **(a)** The code of creating an object (`list`) of `linkedlist` type. **(b)** The runtime object graph of the object `list`. It contains nodes representing the objects and edges the field reference relationship between objects. **(c)** The abstract heap graph of `list`. Each cell represents a point-to set which contains anodes. **(d)** The tagged abstract heap graph, in which the tags are computed to represent the modification of a hypothetical delta region.

demarcated by a persisting call pair $(\widetilde{\xi_p^a}, \widetilde{\xi_p^b})$, we first compute its modification information, extract the abstract heap graph, and then map the modification information to the AHG following Rule 1.

Rule 1. *For any cell, U, in the AHG, if there is a contained anode whose field* `f` *is detected as modified in $delta(\widetilde{\xi_p^a}, \widetilde{\xi_p^b})$, the outgoing edge labeled* `f` *of U should be tagged as* Modification-exclusive *(*MOD-X*)* [9]*. Also, the edges on the paths from root b to U should be tagged as* Modification-related *(*MOD-R*). We refer to them as* MOD-X *edges and* MOD-R *edges respectively.*

The TAHGs are used essentially as state machines instructing the traversal of the runtime persistence process. The state machine encoded in the TAHG can be expressed as a tuple $\langle \sigma, S, \omega, \delta \rangle$, where σ is a set of fields, S is the set of vertices of the TAHG, $\delta : S \times \sigma \rightarrow S$ is the state-transition function, and ω is a set of persistence operations that store the fields (represented as the edges in OG) into byte sequences [10]. When a field is to be visited in the OG by the persistence process, the corresponding transition (represented as the edges in the TAHG) in the TAHG is triggered and certain persistence operation, which is determined by the type of the transition, is carried out.

The mapping between the types of the transitions and the persistence operations is as follows. (1) When the transition is a MOD-R transition, the persistence operation is to simply store the corresponding field (the field to be visited currently in the OG). (2) When the transition is a MOD-X transition, the persistence operation is to store the corresponding field, and also its referenced subgraph in the OG without triggering the transitions of the state machine. Such persistence operation reflects the scenario that an assignment, such as $c.f = p$, could attach a new chunk of the object heap, of which the root is p, to the object heap rooted at c. As the result of the assignment, all reference chains prefixed with $c.f$ point to different objects. Note that, the transitions transitively following the MOD-X transition are excluded from being triggered, which is also the reason why the transition is named as Modification-exclusive. (3) When the transition is neither a MOD-X transition nor a MOD-R transition, it means that both the corresponding field and the referenced subgraph in the OG are not modified. Accordingly, the persistence operation skips storing the field and the referenced subgraph. As the tailoring opportunities come from such kind of transition, we also refer to this type of unlabeled edges as the contributing edges.

[9]We explain soon why it is postfixed with "exclusive".

[10]For the purpose of illustration, we take the JRE OOS output form as an example.

An exemplary TAHG is shown in Figure 4(d). It is constructed for a hypothetical delta region which modifies the field `color` of nodes in the `list` object and sets the field `colored` of `list`. According to the above rule, the fields `id` and `content` of each node and the field `size` of the list do not need to be stored by the persistence process.

In addition, as described in Rule 2, we can adjust the tags in the TAHG to minimize the transitions to be executed while preserving the tailoring capability.

Rule 2. *For any cell U, if all outgoing edges of U are* MOD-X *edges, we automatically promote the tag of the incoming edge of U from* MOD-R *to* MOD-X.

The rule reduces the number of transitions to execute as the incoming edge of U is promoted to a MOD-X edge and the transitions represented by all the outgoing edges of U are no longer executed (Remind case 2 above). This rule can be applied recursively as, once all the outgoing edges of a cell U are promoted to MOD-X edges, the incoming edge of U is then promoted to a MOD-X edge. As a result of the rule, the overhead of executing the state machine is lowered down.

2) Use-Directed Reduction: The use-directed reduction (UDR) analysis aims at finding the states of a persisted object that are not relevant to the subsequent computation after the object is reloaded into memory. Such information can be encoded into a TAHG to guide the persistence tailoring process.

Similar to the IM analysis, the UDR analysis consists of three steps: the use region analysis, the side-effect analysis, and the AHG tagging, as shown in Figure 7. For each persisting call, we first calculate all of the possible use regions by locating the corresponding reloading operations. We then collect the use information and tag the AHGs correspondingly.

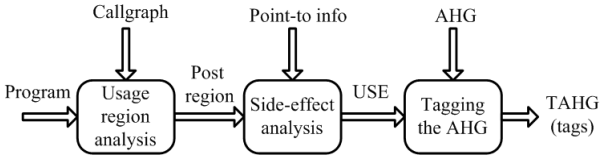


Fig. 7. Analysis phases for use-directed reduction

Use region analysis The purpose of the use region analysis is to statically "predict" what parts of the object being persisted are needed for the future computations with respect to the matching reloading calls. We first formally define the use region as follows.

Definition 7 (Use Region)**.** *Given a persisting call, $\widetilde{\xi}_p^a$, the set of matching reloading calls are those calls, $\widetilde{\xi}_r^b$, that satisfy two conditions: 1. a and b are type compatible by either sub-typing or explicit type cast. 2. The reloading calls are in the post-region of the persisting call that targets a, i.e., $\widetilde{\xi}_r^b \in post(\widetilde{\xi}_p^a)$.*

The use region of ξ_p^a, $use(\widetilde{\xi}_p^a)$, is defined as: $use(\widetilde{\xi}_p^a) := \bigcup_{\widetilde{\xi}_r^i \in \widetilde{R}} post(\widetilde{\xi}_r^i)$.

Following Definition 7, we first capture those reloading calls, $\widetilde{\xi}_r^b$, that match the persisting call, $\widetilde{\xi}_p^a$. We then calculate the post-region of $\widetilde{\xi}_r^b$ and merge all of the post-regions together as the use region of $\widetilde{\xi}_p^a$. In some applications, the reloading call and the persisting call are in different programs that carry out separate stages of some complex computations. For these applications, the use regions of the persisting calls can be analyzed similarly. However, we require the users of Pert to provide the matching specifications for the persisting and reloading operations.

Side-effect analysis and AHG tagging The analysis of the use information is identical to that of the previous analysis, where we inspect every statement in the use region and mark the corresponding edge in AHG if the field is used. For the AHG rooted at the persisted object a, we tag $AHG(a)$ in the following way:

Rule 3. *For each cell, U, of $AHG(a)$, if there is an anode in U whose field* `f` *is used in $use(\widetilde{\xi}_p^a)$, then the outgoing edge labeled* `f` *of U should be tagged as* USE-related *(*USE-R*).*

With the USE-R tags, the TAHG acts as a state machine which helps tailoring the persistence. The interplay between the state machine and the OG traversal is identical to the one described above in the IM analysis, while the mapping between the types of the transitions and the persistence operations is slightly different. When the transition is a USE-R transition, the persistence operation is to simply persist the current field. When the transition is a transition with no tags, the persistence operation is to skip persisting the field and the referenced subgraph in the OG.

In addition, we can adjust the tags in the TAHG according to Rule4, to minimize the transitions to be executed while preserving the tailoring capability.

Rule 4. *For any cell U, if all* transitive *outgoing edges of U are* USE-R *edges, we automatically set the tag of the incoming edge of U to* USE-exclusive *(*USE-X*).*

USE-X is a new kind of tag generated by Rule 4, which means that all the edges transitively following the USE-X edge are USE-R edges. As all the edges following the USE-X edge are USE-R edges, their corresponding fields should be persisted along with triggering the transitions of the state machine (Remind case 1 above). Equivalently, we can achieve the same effect more efficiently by using the USE-X tag. When the transition is a USE-X transition, the persistence operation is to store the corresponding field and the referenced subgraph in the OG without triggering the transitions of the state machine.

C. Limitation

Our IM and UDR algorithms have a few limitations. First, as with other persistence approaches, our persisted object states can become stale if code version changes. That is, if the class definition or the object usage changes as a result of program upgrade or maintenance, the object can not be correctly reloaded. In these cases, we can use engineering solutions such as simply throwing a versioning exception similar to the approach taken by the Java OOS library. But in general, programmers need to realize that code changes invalidate the persisted data from the older versions. Second, our approach currently only optimizes the single-threaded program. To support the multi-threaded program, we need to redefine the delta region and the use region. Take the delta region for example, in multi-threaded program, it should not only contain the intra-thread code between two persisting calls, but also contain the code in other threads which may modify the object states concurrently. We believe that persistence in the multi-threaded programs is a challenging and a separate research topic, left for our future research to address. Finally, our approach can not handle the applications using the Java reflection. Tools such as TamiFlex [1] can be used to mitigate this problem.

III. Implementation

The Pert framework, as illustrated in Figure 8, consists of basic persistence APIs, program analysis tools, and a runtime library. The Pert persisting and reloading APIs can be used as the drop-in replacements of the conventional persistence APIs, e.g., the JRE OOS library APIs. The Pert static analysis is carried out by the *tag generator*, which generates the TAHGs used by the Pert runtime to filter out unnecessary object states during the persistence process. Our main implementation objective is to provide persistence tailoring with low runtime overhead and, at the same time, to ensure the correctness of the tailored persistence. In the rest of the section, we discuss the Pert components in detail and explain how these objectives are achieved [11]. Note that, we mainly adopt the the JRE OOS library as the back end persistence mechanism in our evaluation, so we may discuss our implementation issues based on the JRE OOS library.

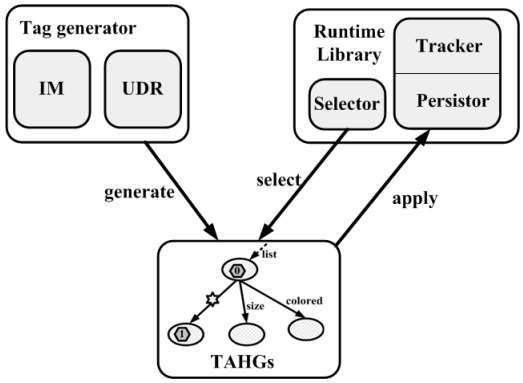


Fig. 8. The architecture of PERT

A. Tag Generator

The Pert *tag generator* statically analyzes the program bytecode, using both the IM and the UDR analysis, and encodes the analysis results as the tagged abstract heap graphs (TAHGs). The TAHGs are constructed using the Spark pointer analysis [8] of the Soot framework [12] by first computing the point-to set, $pt(r)$,

[11] We make our code publicly accessible:http://www.cse.ust.hk/prism/pert

[12] Soot framework. URL:http://www.sable.mcgill.ca/soot/

of the variable r that represents the object to be persisted. $pt(r)$ gives us the anodes denoting the possible allocation sites of r. Following the class definition of r, we compute the point-to sets for its fields of the reference type and repeat the same process for objects pointed to by these field references. The analysis results, encoded as the tags (i.e., MOD-X, MOD-R, USE-X or USE-R), are stored as extra bits in the data structures that represent the TAHG cells. The tag generator is also responsible for indexing the generated TAHGs so that they can be efficiently looked up by the Pert runtime. The TAHG produced by the UDR analysis is indexed by the corresponding persisting call suggesting the start point of the use region. The TAHG of the IM analysis is indexed by two persisting calls demarcating the delta region. If a TAHG encodes the synthesized result with tags from both the IM and UDR, two, instead of three, persisting call chains are needed because the ending call chain of the delta region also acts as the call chain suggesting the start point of the use region.

Specifically, we have two techniques of the tag generator to improve the overall performance of Pert. One is to detect the contributing edges which incur few tailoring capability but high overhead (low-utility edges), the other is to index the TAHGs by the persisting call chains efficiently. We discuss them as follows.

1) Low-utility Edge Detection: As mentioned in Section II-B, when the persistence process executes the contributing transitions in the TAHG, the corresponding field and referenced subgraph in the OG are skipped by the persistence process (benefit). But before the contributing edge is visited in the TAHG, the edges on the leading paths from the root to it are tracked, which brings extra tracking cost (cost). In some applications, there may exist a fraction of contributing edges which incur high tracking cost while introducing low benefit. We refer to such contributing edges as low-utility edges. For low-utility edges, we detect them through a static cost-benefit model and discard the tailoring opportunities to avoid the unworthy high tracking cost. To discard the opportunities, we treat the low-utility edges conservatively as MOD-X (or USE-R) edges and automatically apply Rule 2 (or Rule 4) to reduce transitions to be executed. We first show a cost-benefit model, then the detection strategy based on it.

In general, we use the number of fields to statically approximate the cost and benefit. As for the tracking cost of a contributing edge, it is proportional to the number of fields on the path which leads the persistence process from the root to the edge. Suppose the leading path of a contributing edge is a sequence of fields: [f_1, f_2, …, f_n], the number of fields, n, could be an approximation of the tracking cost. In addition, a field may be shared by multiple paths and the cost of the field should be better amortized. We adjust to approximate the cost using $\sum_{i=1}^{n} 1/cn(f_i)$ where function cn returns the number of children of f_i (for the array, we assume it has 100 children by default). As for the benefit of the a contributing edge, it equals the number of fields tailored at runtime. We statically approximate it using the transitive size of the TAHG edges under the contributing edge. With the above model, we judge a contributing edge as low-utility edge if its cost is larger than its benefit. For those low-utility edges, we discard the corresponding tailoring opportunities to avoid the high tracking cost.

2) Indexing the Generated TAHGs: A persisting call $\widetilde{\xi_p^o}$ is defined as a call chain: $\xi_1 \rightsquigarrow \cdots \rightsquigarrow \xi_n \rightsquigarrow \xi_p^o$. We adopt a simple but efficient hashing scheme [2] to encode the call chain, i.e., the persisting call. It maps a call chain to an *unique probabilistic value*. In particular, the hash value (I) of $\widetilde{\xi_p^o}$ is defined in the following equation. Note that g is a function assigning a predefined `int` value to each call site. "$\times$" is the multiplication (modulo 2^{32}) operation, "+" the addition operation (modulo 2^{32}), and the range of I is $[0 \ldots 2^{32}-1]$. While simple, such hashing function exhibits very low conflict ratio, and thus enables fast retrieval.

$$\begin{array}{lcl} I(\widetilde{\xi_p^o}) & = & 3 \times I_{\xi_1 \rightsquigarrow \cdots \rightsquigarrow \xi_n} + g(\xi_p^o) \\ I_{\xi_1 \rightsquigarrow \cdots \rightsquigarrow \xi_n} & = & 3 \times I_{\xi_1 \rightsquigarrow \cdots \rightsquigarrow \xi_{n-1}} + g(\xi_n) \\ I_{\xi_1 \rightsquigarrow \xi_2} & = & 3 \times I_{\xi_1} + g(\xi_2) \\ I_{\xi_1} & = & g(\xi_1) \end{array} \quad (1)$$

B. The Pert Runtime

The Pert runtime consists of the two components: the selector and the serializer. The selector looks up the TAHG, using the current execution context of the persisting call. The serializer uses the TAHG to tailor the persistence so that only interesting objects states are serialized into byte arrays, which are later written to the disk. To better understand the mechanism of the Pert runtime, let us first review how the conventional serialization of a Java object works. Each Java class contains the *metadata* describing its fields. Using these *metadata*, a conventional persistence process traverses the object graph of the persisted object. It first stores the `metadata`, then the primitive fields, and last the field references, into byte sequences. The traversal process is then repeated for each of the objects pointed to by the field references. The main objective of the Pert runtime is to use the information stored in the TAHGs to skip the unnecessary serialization operations during the traversal.

1) Selector: The main functionality of the selector is to find the proper TAHGs to assist current persistence call. Remind that TAHGs are indexed by call chains. For the UDR analysis, the current persisting call can acts as the searching key. But for the IM analysis, an earlier persisting call chain is needed because the TAHGs represent a delta region bound by two call chains. The detailed workflow of selector is as follows. At runtime, before the Pert persistence API is executed, the selector first identifies the current calling context by walking the stack via the `Thread.getStackTrace` [13] API. If the IM analysis is enabled, the selector continues to search in a cache for an earlier call chain that persists the same object. This cache is maintained by the selector to store the persisted objects and their most recent persisting call chain.

2) Serializer: The input of the Pert serializer includes both the object to be persisted and the selected TAHG. The serializer then recursively traverses the object graph using the class meta-data, obtained by the Java runtime reflection facilities, and simultaneously writes the state information into a byte array. For each step of the traversal of the OG, the corresponding edge in the TAHG is also visited. If the edge in the TAHG has no tags associated with it, which means that the corresponding field is not modified incrementally or to be used in the subsequent computation, then the field is skipped from the persistence and so is the referenced subgraph in the OG.

To lower the runtime overhead of the serializer, we have also implemented a few optimization techniques.

[13] http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html

Treating runtime aliases Multiple alias variables may reference the same object at runtime, to avoid the same object being persisted multiple times by one persistence call, the serializer maintains a list of the object identifiers of the persisted objects during the traversal of an object graph and only persist the identifier itself if the object has been persisted.

Compression before the externalization If the byte arrays are large, we compress them before writing to disk. The serializer uses multiple threads to overlay the compression operation, a CPU-bound task, with the disk writing tasks. One interesting observation is that persisting with two threads is about 2X faster than use of one thread. However, more threads do not introduce any further improvements. After investigation, we find that the bus reaches its bottleneck before the CPU, at the time that two threads are present. Note that the compression technique is a specific optimization for the object serialization mechanism, which is not applicable to the object-relational persistence mechanism.

Hash code problem The default hash code implementation for Java objects allows the hash code of the same object be different across different runs. The underlying reason is that the default hash code depends on the internal address of the object in a particular JVM instance, and the object may not have the same address after being swapped to and from the disk. This brings problems for persisting data structures that are dependent on the object hash code, such as `HashMap`. To solve this problem, the JRE OOS library adopts a mechanism that the key objects and the value objects are persisted separately to neighboring disk addresses. This "neighborhood"-based scheme, subsuming the runtime mapping information between *key* and *value*, can be used to recover a map entry by inserting the *key object* and *value object* into a newly created `HashMap` instance. This *reconstruction*-based reloading mechanism, as shown in the Table V, incurs the significant runtime cost.

In Pert, a special `hashCode` method is provided for those classes inheriting the default `hashCode` method of the `Object` class. We insert a static field *logical_id* in the `Object` class. The ID is increased every time when the constructor of the `Object` class is invoked. By this scheme, the hash code remains the same after an object is swapped out of and into the memory. For dynamically loaded classes, we apply the load-time transformations through bytecode engineering.

IV. Evaluation

To evaluate Pert, we conduct an extensive set of experiments, using third party Java programs, to quantify the effect of the persistence tailoring with respect to the program speedup, the reduction of the persisted data, and the overhead of the tailoring process itself. For collecting these metrics, we selected object serialization as the back end persistence mechanism. We also conducted a study to show our optimization is applicable to other back end mechanism, e.g., object-relational persistence.

A. Experimental setup

We list the subject programs used in our evaluation in Table I. For each program, we show the number of application classes reachable from the class of the `main` method ($Classes$), the methods ($Methods$), the statements in Soot's internal representation ($Stmts$), the lines of java code excluding blank lines (LOC), the instance fields and static fields connected by the symbol "+" ($Fields$). A difficulty in our evaluation is that, to the best of our knowledge, there are no standard benchmarks for object persistence operations. Therefore, we first picked the subjects that are also used in the evaluation of the checkpointing technique [19], a technique making heavy use of object persistence. specjbb2005 [14] is a business transaction benchmark, configured to run as a single-threaded program. jgrapht [15] is a free Java graph library that provides mathematical graph-theory objects and algorithms. We use the example code distributed with the library. The technique being compared is the Pert runtime library with optimization disabled. It is equivalent to the functionality of conventional persistence libraries such as the JRE OOS facilities.

$Program$	$Classes$	$Methods$	$Stmts$	LOC	$Fields$
compress	22	163	2369	506	52+54
jess	117	673	9959	6784	215 +100
jflex	78	609	14577	12679	274+223
jgrapht	23	264	3680	3366	89+59
jtar-1.21	8	51	1198	601	29+22
muffine-0.9.3	309	2002	43412	26609	754+316
raytrace	40	337	4996	4670	109+60
sablecc	449	3515	30605	29753	690+321
socksecho	34	273	6646	3805	141+82
socksproxy	23	264	3680	3366	89+59
soot-2.2.3	2556	16239	309083	182673	3671+1000
specjbb2005	82	680	17466	12757	556+286
violet	324	1364	25242	15433	461+138

TABLE I
Analyzed programs

For the selected subjects, we carefully insert persistence operations into the benchmarks as follows. To evaluate the incremental modification analysis, we inject two persistence calls at the start and the end points of transactional regions. For the benchmark soot, we select the pack, a basic business unit, as the transactional region. For other benchmarks that do not use transactions, we identify the operations that use large data structures, under the assumption that the programmer could be motivated to persist the data structure to avoid the repetitive and lengthy initialization. To evaluate the use-directed reduction, we simulate the technique of checkpointing as described by Xu et al. [19] as follows. When the program reaches a pre-defined checkpoint, we first persist the local variables and static variables used in the rest of the execution. We then reload them directly and continue the execution. Pert, through modeling the heap statically and tagging it with the side effects, persists only interesting parts of the OG referenced by the local variables or static fields. All experiments are conducted on a 4-core Intel CPU running 2.6 Linux kernels using the JRockit R27 32-bit JVM. Each data point is an average of ten identical runs, and the value of each run is within the small range $[\mu - \varepsilon, \mu + \varepsilon]$ where μ equals to the expected value of the variable observed in ten runs, ε is the standard deviation [16].

B. Study of Incremental Modification

This study aims at quantifying the speedup and size reduction as the result of performing the incremental modification analysis. For each subject program, we show in Table II the subject program ($Program$), the time (ms) of original persisting

[14] http://www.spec.org/jbb2005/
[15] http://jgrapht.sourceforge.net/

and reloading operation ($Orig_p$ and $Orig_r$), the time (ms) of optimized persisting and reloading operations (Opt_p and Opt_r), the corresponding speedup ($Impr_p$ and $Impr_p$), the size (byte) of the persisted data before ($Orig_{size}$) and after (Opt_{size}) the optimization, and the ratio of the size reduction ($Redu_{size}$).

The data shows that our method is effective in eliminating redundant persistence work. Taking the subject `compress` as an example, the `run_compress` method compresses the data stored in the field `orig_text_buffer` into the field `comp_text_buffer`. Pert detects that the field, `orig_text_buffer`, is not modified during the compression operation. Therefore, the second persistence operation performed after the compression can skip storing the unmodified field. For this subject, the unoptimized persisting durations for the the first and the second operations are 20ms and 20ms, whereas the reloading durations are 16ms and 18ms respectively. With Pert, the second persistence call turns out to be 10 ms, a 2X speedup, and the second reloading operation 4ms, an over 4X speedup. The persisted data size is also half of the conventional approach: a reduction to 3.154M bytes from 6.308M bytes.

In general, the incremental modification analysis is very useful in reducing the amount of persisted data and speeding up the persistence operation. For the subjects except jgrapht, jtar and violet, our experiments show that the reduction ratio of persisted data size ranges from 6.7% to 99.9%, where in the cases of jess, jflex, muffine, soot and specjbb, the ratios are even higher than 90%. These five subject programs greatly benefit from Pert because their transactional units often only mutate, through a long-lasting computation, a small part of a large data structure. The subjects jtar and jgrapht experienced slight slowdown and the data reduction is 0%. As for jtar, all the fields are modified. As for jgrapht, it makes heavy use of a single collection for which the static analysis can not distinguish the updated indices from those that are unchanged. The conservative result suggests all the indices to be persisted, as well as the objects referenced by indices. The slowdown comes from the runtime overhead of the TAHG traversal. The slowdown is low as a result of Rule 2: When the children edges are all MOD-X edges, the rule can replace the tracking of children edges to the tracking of their parent edge, and also such rule can be applied recursively, hence, reduces the number and the cost of tracking operations.

The mapping between the size reduction and the performance improvement is not a monotone function. As seen from Table II, compared to the benchmark raytrace, the benchmark socksecho has a smaller size reduction ratio but has a higher performance improvement ratio. An investigation into the implementation and the runtime execution reveals that, besides the size of the objects skipped during the persistence process, the type of the objects skipped also affects the performance. In particular, the class meta-data is persisted along with the object. Reducing the objects to be persisted (through skipping the referencing field) implicitly reduces the class meta-data to be persisted. Also, the class meta-data is resolved through the Java reflection, a costly operation. Reducing the objects to be persisted may implicitly reduce the resolution of some classes which are not used by the persistence process any more. As the complexity of the classes differs from one benchmark to the other, we can not expect the performance improvement to be a linear function of the size reduction. However, a general observation is, when the data to be persisted or reloaded is reduced, the total time spent on the persisting or reloading is also reduced.

C. Study on Use-Directed Reduction

This study investigates the effect on performance improvement and data reduction of the use-directed reduction analysis. The results are reported in Table III, of which the columns are interpreted in the same way as in Table II. In addition, we compare the performance of our UDR approach with that of the state of the art approach proposed by Elbaum et al. [4].

We observe from Table III that the UDR analysis is also effective in the speedup of the persisting/reloading operation, as well as the reduction of the persisted data size. Overall, the reduction of persisted data size ranges from 6.7% to 91.3%. The UDR analysis works well for cases such as soot, where some fields are not used later and can be filtered out from the persistence. As as result, the data persisted is reduced by 91.3% compared to the unoptimized version. In the cases of jgrapht, muffine, raytrace and violet, the use region accesses all fields. Therefore, no field is skipped from the persistence and the size of the persisted data is not reduced. The slowdown is low as a result of Rule 4.

To compare with the approach of Elbaum et al. [4], we faithfully implemented their approach (*May-reference reachable projection*). Since their original approach supports only the use region bounded by the method entry and method exit points, we provided a non-trivial extension to support the general form of use region delimited by any call chains. The performance speedup of the persisting and the reloading is shown in Figure 9(a) and Figure 9(b), respectively. As shown in both figures, for all the benchmarks, the speedup incurred by our approach is higher than the speedup incurred by their approach. For the benchmarks jgrapht, muffine, sablecc, soot and violet, the speedup incurred by our approach is even 20% higher on average.

The underlying reason for the performance difference is the use of TAHGs in our approach. In particular, our approach marks the used field with the tag on the TAHG and determines whether to persist a field by checking the existence of the tag. Without TAHG, their approach marks the used field using the reference chain, e.g., `o.field1.field2`. During the traversal of each field in the OG, their approach checks the current reference chain against the set of pre-defined ones to make the persistence decision. Such checking is essentially implemented as the pattern matching, which is more costly compared to simply checking locally the existence of the tag in our approach. In addition, checking reference chains that share the common prefix may unnecessarily duplicate the task of pattern matching. For instance, checking the reference chains `o.field1.field2` and `o.field1.field3` both drive the state machine for pattern matching, from the initial state to a specific state with the prefix `o.field1`, an action avoided in our approach. What is more, their approach, due to its encoding nature, can not easily support the adjustment of tags (Rule 2 and Rule 4 in Section II-B), which can greatly reduce the matching overhead. The above drawbacks of their approach manifest themselves when the object graph of the object to be persisted is complex. In the benchmarks jgrapht, muffine, sablecc, soot and violet, the object graph of the object to be persisted is often complex and the overhead of checking the reference chain is not ignorable, as the result of the above drawbacks of the checking, their approach does not perform well compared to our approach.

$Program$	$Orig_p$	Opt_p	$Impr_p$	$Orig_r$	Opt_r	$Impr_r$	$Orig_{size}$	Opt_{size}	$Redu_{size}$
compress	21.1	11.2	46.9%	19.1	5.3	72.3%	6,308,014.0	3,154,007.0	50.0%
jess	20.2	1.2	94.1%	30.0	0.8	97.3%	117,915.0	3,065.8	97.4%
jflex	40.8	5.2	87.3%	72.5	4.7	93.5%	610,444	308	99.9%
jgrapht	52.7	53.0	-0.6%	61.7	63.5	-1.3%	33,906.0	33,906.0	0.0%
jtar-1.21	11.1	11.2	-0.9%	12.3	12.5	-1.6%	11,597	11,597	0.0%
muffine-0.9.3	12.1	1.9	84.3%	8.7	0.9	89.7%	54,952	1,209	97.8%
raytrace	4.9	2.5	45.0%	5.0	2.6	48.0%	345.0	138.0	60.0%
sablecc	81.7	70.5	13.7%	91.3	79.2	13.3%	522,872	487,810	6.7%
socksecho	8.6	3.9	54.7%	10.3	4.6	55.3%	2,021	1,466	27.5%
socksproxy	24.8	11.4	54.0%	22.6	15.2	32.7%	332,724	300,105	9.8%
soot-2.2.3	102,813.3	29,480.4	71.3%	112,589.0	300,15.2	73.3%	984,556,231	86,110,672	91.3%
specjbb2005	1,532.0	1.4	99.8%	1,580.0	2.9	99.8%	10,829,910.0	10,829.9	99.9%
violet	91.3	85.1	6.8%	96.6	87.5	9.4%	822,688	800,136	2.7%

TABLE II
IMPROVEMENT BROUGHT BY THE INCREMENTAL MODIFICATION ANALYSIS

$Program$	$Orig_p$	Opt_p	$Impr_p$	$Orig_r$	Opt_r	$Impr_r$	$Orig_{size}$	Opt_{size}	$Redu_{size}$
compress	23.2	11.7	49.6%	21.2	6.1	71.2%	6,308,014.0	3,154,007.0	50.0%
jess	18.3	16.5	9.8%	22.4	20.6	8.0%	117,915.0	102,586.1	13.0%
jflex	49.7	42.8	13.9%	67.0	60.1	10.3%	610,444	569,536	6.7%
jgrapht	53.9	54.6	-1.3%	63.2	64.5	-2.1%	33,906.0	33,906.0	0.0%
jtar-1.21	11.5	9.3	19.1%	12.6	9.5	24.6%	11,597	10,253	11.6%
muffine-0.9.3	11.9	12.3	-3.3%	9.8	10.3	-5.1%	54,952	54,952	0.0%
raytrace	5.0	5.3	-5.0%	7.0	7.2	-2.9%	345.0	345.0	0.0%
sablecc	82.3	78.8	4.3%	89.9	86.8	3.4%	522,872	510,810	2.3%
socksecho	9.2	3.7	59.8%	11.0	5.1	53.6%	2,021	1,466	27.5%
socksproxy	22.6	13.1	42.0%	20.6	14.2	31.1%	332,724	300,105	9.8%
soot-2.2.3	103,427.8	30,568.4	70.5%	106,597.3	31,213.9	70.7%	984,556,231	86,110,672	91.3%
specjbb2005	1,790.0	1678.2	6.2%	1,557.6	1421.5	8.7%	10,829,910.0	9,649,449.8	10.9%
violet	90.9	90.1	0.8%	97.5	98.2	-0.7%	822,688	822,688	0.0%

TABLE III
IMPROVEMENT BROUGHT BY THE USE-DIRECTED REDUCTION ANALYSIS

D. Study on Overall Effect

In Table IV, we show the overall improvement with both the IM and UDR analysis enabled. The method entry and exit points are the start and the end points of the delta regions and the end point also acts as the persisting call determining the use region. The Table IV shows the performance improvement and the size reduction of the Pert-based persistence at the end point incurred by both the IM and the UDR analysis.

Comparing the data in Table IV to Table II, we make an interesting observation that most cases do not enjoy further improvement from the UDR analysis after the IM analysis is used. This is because those fields modified previously in the delta region are almost always used later in the use region. One exception is in the benchmark jtar, where the IM analysis reduces no data but the UDR analysis reduces the size of persisted data by 11.6%. A closer inspection shows the underlying reason: the object to be persisted hosts the same copy of data in two parts, one is in its array field while the other is in its referenced `OutputStream` object, two parts are both updated while only one part is used later.

E. Low-utility Edge Detection

As mentioned in Section III-A, we detect the low-utility contributing edges which incur the high tracking cost but introduce low benefit, for which the corresponding tailoring opportunities are discarded for better performance. To quantify the effect of the low-utility edge detection, we examine the performance in the evaluation of the incremental modification analysis, with and without the low-utility edge detection. The comparison is shown in Figure 10. From the figure, we see that the performance speedup of applications with complex data structures often increases due to the low-utility edge detection. For example, in subjects sable, soot, violet, the performance speedup with the low-utility edge detection is around 10% higher than without the detection. However, the effect of low-utility edge detection on small benchmarks is ignorable. The reason is that, in simple object graph, even if the low-utility edges exist according to our cost-benefit model, their costs are actually ignorable because the leading paths are short.

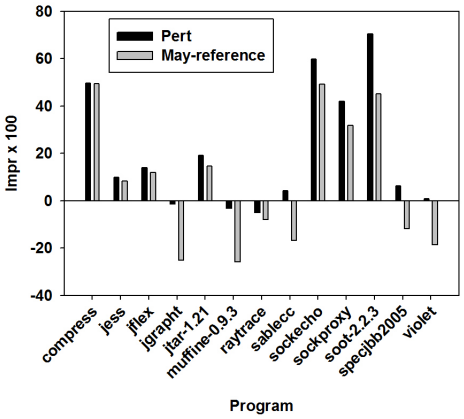
F. Tailoring of Object-relational Persistence

In this section, we show that our algorithm is applicable to other persistence mechanisms, e.g., object-relational persistence. The object-relational persistence is a main stream persistence mechanism that stores the objects to relational database instead of using a serial alignment. Frameworks of this persistence style, such as Hibernate, are widely deployed. Without losing generality, we apply our IM algorithm to the extended OO7 benchmark [3] [16]

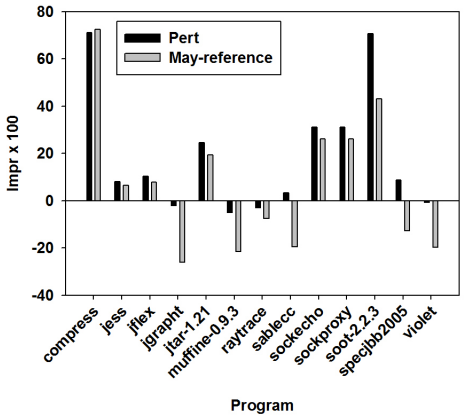
[16]The OO7 benchmark is extended to contain Hibernate operations. URL: http://oo7j.sourceforge.net/

$Program$	$Orig_p$	Opt_p	$Impr_p$	$Orig_r$	Opt_r	$Impr_r$	$Orig_{size}$	Opt_{size}	$Redu_{size}$
compress	25.6	11.9	53.5%	24.1	7.8	67.6%	63,08,014.0	3,154,007.0	50.0%
jess	23.2	0.8	97.0%	24.7	1.1	96.0%	117,915.0	3,065.8	97.4%
jflex	44.7	1.6	96.4%	66.8	4.7	93.0%	610,444	308	99.9%
jgrapht	53.2	54.1	-1.7%	63.6	65.0	-2.2%	33,906.0	33,906.0	0.0%
jtar-1.21	11.8	9.7	17.8%	12.7	9.3	26.8%	11,597	10,253	11.6%
muffine-0.9.3	13.1	1.9	85.5%	9.1	1.0	89.0%	54,952	1,209	97.8%
raytrace	6.8	3.2	52.9%	6.5	4.2	35.4%	345.0	138.0	60.0%
sablecc	82.1	71.3	13.2%	91.6	80.1	12.6%	522,872	487,810	6.7%
socksecho	8.8	3.8	56.8%	10.7	4.6	57.0%	2,021	1,466	27.5%
socksproxy	23.4	12.3	47.4%	22.4	13.7	38.8%	332,724	300,105	9.8
soot-2.2.3	111,026.6	28,779.3	74.1%	108,617.4	30,463.1	72.0%	984,556,231	86,110,672	91.3%
specjbb2005	1,670.0	2.6	99.8%	1,522.0	1.9	99.9%	10,829,910.0	10,829.9	99.9%
violet	92.1	85.8	6.8%	97.4	89.9	7.7%	822,688	800,136	2.7%

TABLE IV
OVERALL IMPROVEMENT

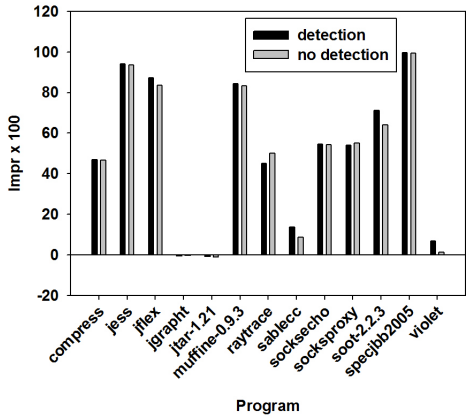


(a)

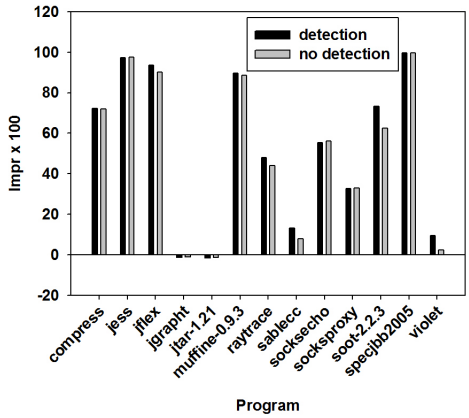


(b)

Fig. 9. **(a)** Comparison of the improvement of persisting time, the left bar represents our approach, and the right bar represents the approach of Elbaum et al. **(b)** Comparison of the improvement of reloading time. The meaning of the left bar and right bar is the same as in (a).



(a)



(b)

Fig. 10. **(a)** Comparison of the improvement of persisting time, the left bar represents the version where the low-utility edge detection is applied, and the right bar stands for the version without the detection applied. **(b)** Comparison of the improvement of reloading time. The meaning of the left bar and right bar is the same as in (a).

which persists the runtime states with Hibernate. The transaction that we optimize retrieves the `BaseAssembly` objects from the database, updates their `componentsPrivate` fields and then stores them back to the database. Hibernate relies on a `Persister` object that creates the template for the JDBC update statement, invokes the runtime-generated `getter` methods to get the values of fields, and instantiates the template to executable JDBC update statement. We modify the Hibernate code so that it is aware of the application's behavior and only persists the modified fields since last update or retrieval.

As the table in the database has a fixed scheme, the size of a table is not a good measurement of the size reduction. Instead, we use the size of updated fields in the table to measure

the size reduction during the persistence. When measuring the performance speedup, we turn off other Hibernate optimization, e.g., caching, to isolate the effect of our algorithm. Our results show that, the size of updated fields is reduced from 4417 KB to 968KB, the time for incrementally persisting the objects is reduced from 33,070 ms to 26,178 ms, and the time for the reloading is reduced from 35,129 ms to 27,720 ms. As seen from the data, the size of updated fields is reduced by 78% and the time for the persisting or the reloading is reduced by around 20%. From this study, we see that our algorithm is orthogonal to the back-end persistence mechanism, which is consistent with the fact that, the algorithm decides what data to persist is independent of the persistence mechanism decides how to persist the selected data.

G. The Assessment of Additional Optimizations

We have also evaluated the effects of the optimization techniques adopted in our implementation, including the concurrent compression technique and the customized object identification scheme.

Compression As mentioned in Section III, we use a concurrent compression operation before persisting data to disk to improve the bus throughput and to make better use of the CPU. To assess this technique, we conduct a controlled experiment, where the objects are persisted and reloaded multiple times with and without the compression applied respectively. We utilize the `Pseudograph` type from jgrapht benchmark to instantiate two graph objects. The compression effect for two graphs is shown in Figure 11 and Figure 12 respectively. Using compression, the data is first compressed using `GZIPOutputStream` [17], then the chunk of compressed data is dumped to the disk. The corresponding reloading call works in the reverse order where the chunk of compressed data is reloaded to memory and then decompressed. Obviously, the "persisting with compression" and "reloading with decompression" should work in pairs. Figure 11 and Figure 12 shows the time consumed for four cases: persisting with compression, persisting without compression, reloading with compression, reloading without the compression. The time is the total time consumed by the persistence operation and it includes the time spent on writing to the hard disk storage. From either figure, we know the compression can reduce the persisting/reloading time by around 50%.

Object identity As presented in Section III, we implemented a different object identity scheme as a replacement of the default JRE implementation to improve the persistence performance and to preserve the program correctness at the same time. To evaluate these properties, we measure the durations of both persisting and reloading a `HashMap` object, of which the keys and the values are both of the `Object` type. We insert a *key object* and *value object* pair multiple times into the map. To validate the correctness of the hash code, we verify that, given the key object, if the value returned by the reloaded `Hashmap` instance is the same value object as the one used for initializing the map. To evaluate performance, we compare the time (ms) for the persisting and reloading operations using the JRE approach (p_{jre}/r_{jre}) to the time (ms) of Pert-based persisting and reloading operations (p_{pert}/r_{pert}), as the number of entries ($size$) of the map increases. The results

[17]GZIPOutputStream. URL:http://java.sun.com/j2se/1.5.0/docs/api/java/util/zip/GZIPOutputStream.html

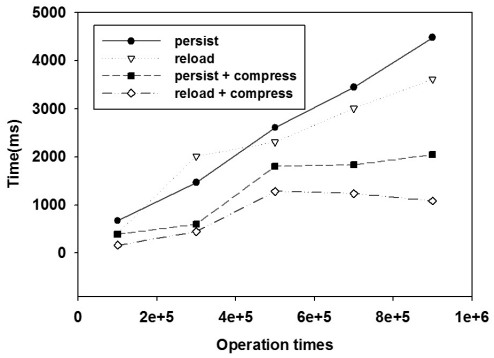


Fig. 11. The effect of compression (graph1)

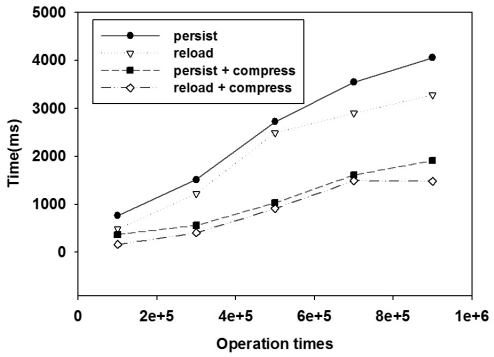


Fig. 12. The effect of compression (graph2)

in Table V show that our strategy not only correctly supports the map function but also outperforms the JRE implementation while the number of objects in the `HashMap` is large. When the map has less than 100 entries, the comparative performance slowdown of the JRE approach is not obvious. Both the persisting and the reloading time (p_{jre} and r_{jre}) taken by the JRE approach increase linearly with respect to the number of map entries. The time taken by Pert, however, stays constant because we do not need to repopulate the map as otherwise required in the JRE case.

V. Related Work

The research that optimizes the persistence process can be classified into three categories: to improve the internal representation of persisted data, to batch the persistence operations, and to make the persistence aware of the applications.

$size$	p_{jre}	p_{pert}	r_{jre}	r_{pert}
10	14	12	6	3
100	19	12	14	3
1000	140	14	143	3
10000	508	12	498	3
50000	1387	12	1186	5
100000	2360	12	1981	3
1000000	22428	16	18482	3

TABLE V
Different treatments for the hash code problem

The first category [12], [13] improves the internal representation of persisted data such as hardcoding the class layerout information and using more compact representation of the class metadata. Philippsen et al. [13] presented one of the earliest works that uses the explicit marshaling instead of the reflection mechanism. Similar techniques are also used in persistence frameworks such as Protobuf and Jackson. Opyrhcal et al. [12] optimizes the persistence process by compressing the common class descriptors, observing that the descriptors in Java often share a long common prefix, such as the common package name. The compression operation of Pert can also achieve this effect. But, more importantly, these approaches do not leverage the application contexts of the persistence operations.

Batch processing [6], [18] is another direction of optimization by observing that, if we need to query data multiple times in a program, it is more efficient to batch the operations in one query. Wiedermann et al. [18] converts the scattered queries written in Java APIs to one equivalent query. Their approach collects information such as the relevant fields and tables from those separate queries, retrieves the filtering conditions from the program paths, then synthesizes one equivalent HQL [18] statement. Their approach is orthogonal to ours because they inspect whether or not the operations should be carried out incrementally or in a batch and do not examine intra-object states. Autofetch [6] follows a similar heuristic: it profiles the query history and predicts the future access patterns. It is a dynamic approach, to which our work is complementary.

The third category of optimization techniques [4], more akin to our approach, use program analysis techniques to examine the application contexts of persistence operations. As also discussed in the introduction, Elbaum et al. [4] inspect statically how the objects are used after they are reloaded. And only the used parts of such objects are persisted and reloaded. But their approach is specially designed for unit testing purposes, where the use region is simply delimited within an unit test method. Their approach does not consider the general use region scenario of a persistence call, and delta regions are not considered at all.

Checkpointing, as mentioned in Section I, is a client technique that uses the object persistence to store the states of application objects to disk. The perspective of checkpointing, exemplified by a state of the art approach proposed by Xu et al [19], determines the necessary set of local variables and static fields for safe replay with respect to the specific application usage scenarios. As far as individual objects are concerned, their persistence operation is an all-or-nothing decision without considering the usage scenarios of the object fields. Our approach can make checkpointing more efficient by automatically deciding the sub-states of an object that are necessary to be persisted. We achieve this through modeling the heap and analyzing the application-specific usage of the heap.

VI. Conclusion and Future work

In this paper, we presented Pert, a framework that provides the tailored object persistence behavior according to the program execution context. Pert makes use of two general observations. First, if multiple persistence operations are issued, we can calculate what has been incrementally modified since the last persistence call. Second, we can examine the future executions of the program with respect to a persistence operation and decide what parts of the object state are actually used, hence, needed to be persisted. Based on these two observations, we designed and developed two optimization algorithms that statically analyze the bytecode of Java programs. The analysis results are encoded in a special date structure, called the tagged abstract heap graph, that is used by the Pert runtime library to make adaptive persistence tailoring decisions. We evaluated Pert on instrumented third-party Java benchmarks and observed dramatic performance improvements as the result of considering the context of the persistence operations. In future, we plan to evaluate the effectiveness of the static analysis by using more sophisticated pointer-analysis techniques [10], [11], [17]. We also plan to support efficient object persistence for multi-threaded programs.

[18]HQL. URL:http://docs.jboss.org/hibernate/core/3.3/reference/en/html/queryhql.html

References

[1] Eric Bodden, Andreas Sewe, Jan Sinschek, Mira Mezini, and Hela Oueslati. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*. ACM, May 2011. To appear.

[2] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA*, pages 97–112, 2007.

[3] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. In *Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 12–21, New York, NY, USA, 1993. ACM.

[4] S. Elbaum, Hui Nee Chin, M.B. Dwyer, and M. Jorde. Carving and replaying differential unit test cases from system test cases. *TSE*, 35(1):29–45, Jan.-Feb. 2009.

[5] Christof Fetzer, Karin Hgstedt, and Pascal Felber. Automatic detection and masking of non-atomic exception handling. *TSE*, 30:2004, 2004.

[6] Ali Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *ECOOP*, 2006.

[7] Tanzima Zerin Islam, Saurabh Bagchi, and Rudolf Eigenmann. Falcon: a system for reliable checkpoint recovery in shared grid environments. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 50:1–50:12, New York, NY, USA, 2009. ACM.

[8] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *CC*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[9] Mark Marron, Manuel V. Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, pages 245–259, 2008.

[10] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for java. *SIGSOFT Softw. Eng. Notes*, 27(4):1–11, 2002.

[11] Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. *PLDI*, 41(6):308–319, 2006.

[12] Lukasz Opyrchal, , Lukasz Opyrchal, and Atul Prakash. Efficient object serialization in java. In *Proceedings of 19th IEEE International Conference on Distributed Computing Systems Workshops*, 1998.

[13] Michael Philippsen and Bernhard Haumacher. More efficient object serialization. In *IPPS/SPDP Workshops*, pages 718–732, 1999.

[14] James Plank, James S. Plank, Micah Beck, Micah Beck, Gerry Kingsley, Gerry Kingsley, Kai Li, and Kai Li. Libckpt: Transparent checkpointing under unix. pages 213–223, 1995.

[15] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for java. In *ASE*, pages 114–123. ACM Press, 2005.

[16] Ronald E. Walpole, Raymond H. Myers, Sharon L. Myers, and Keying Ye. *Probability and Statistics: For Engineers and Scientists*. Pearson Education Inc., Seventh edition, 2006.

[17] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, New York, NY, USA, 2004. ACM.

[18] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA*, 2008.

[19] Guoqing Xu, Atanas Rountev, Yan Tang, and Feng Qin. Efficient checkpointing of java software using context-sensitive capture and replay. In *FSE*, pages 85–94, New York, NY, USA, 2007. ACM.

[20] Andreas Zeller. Yesterday, my program worked. today, it does not. why? In *FSE*, pages 253–267, London, UK, 1999. Springer-Verlag.

Peng Liu received the BEng degree from the University of Science and Technology of China (USTC) in 2008. He is currently working toward the PhD degree in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology. He is interested in optimizing the performance, detecting concurrent bugs and automatically fixing concurrent bugs. In addition, he is interested in the program analysis techniques.

Charles Zhang obtained his Ph.D, M.Sc, and B.Sc. with honors, all from the Department of Electrical and Computer Engineering at University of Toronto. He has published extensively at premium conferences and journals such as IEEE TPDS, OOPSLA, ECOOP, ACM/USENIX MIDDLEWARE, and AOSD. He is also a two-time IBM PhD fellowship winner. Prior to his science endeavor, he spent a year in Beijing Normal University studying history. Before his graduate study, he worked as a software engineer at Motorola and a Silicon Valley startup. He is currently an assistant professor at the Hong Kong University of Science and Technology. His general research interest has been to study how software engineering techniques improve the ways we design, implement, and deploy complex software systems such as distributed systems, middleware, operating systems, and embedded software. His current research focuses on using program analysis techniques, both static and dynamic, to assist programmers in writing concurrent software systems with good quality. He has served as a program committee member for several international conferences, including the International Conference on Software Engineering and the Conference On Object-Oriented Programming Systems, Languages and Applications. He is a member of the IEEE.