Frontmatter

Editors

SIGAPP FY’16 Quarterly Report

J. Hong

Selected Research Articles

Automated Extraction of Business Documentation in Enterprise Information Systems

K. Cemus and T. Cerny

Scheduling Algorithm for Parallel Real-Time Tasks on Multiprocessor Systems

C. Kuo and Y. Lu

Record Data Structures in Racket: Usage Analysis and Optimization

T. Pape, V. Kirilichev, C. Bolz, and R. Hirschfeld

AutoFix: An Automated Approach to Memory Leak Fixing on Value-Flow Slices for C Programs

H. Yan, Y. Sui, S. Chen, and J. Xue
Applied Computing Review

Editor in Chief
Sung Y. Shin

Associate Editors
Hisham Haddad
Jiman Hong
John Kim
Tei-Wei Kuo
Maria Lencastre

Editorial Board Members

Gail-Joon Ahn
Yin-Fu Huang
Anand Paul

Davide Ancona
Angelo Di Iorio
Manuela Pereira

João Araújo
Seiji Isotani
Diego Perez-Palacin

Maurice Ter Beek
Hasan Jamil
Ronald Petric

Giampaolo Bella
Jinman Jung
Peter Pietzuch

Paolo Bellavista
Soon Ki Jung
Beatriz Pontes

Albert Bifet
Christopher D. Kiekintveld
Fernando De la Prieta

Ig Ibert Bittencourt
Bongjae Kim
Rui P. Rocha

Sander Bohte
Dongkyun Kim
Pedro P. Rodrigues

Gloria Bordogna
Sang-Wook Kim
Juan Manuel Corchado Rodriguez

Barrett Bryant
Stefan Kramer
Florian Röhrlbein

Jaelson Castro
S.D Madhu Kumar
Agostinho Rosa

Tomás Cerny
Cecilia Laschi
Davide Rossi

Alvin Chan
Cecilia Laschi
Giovanni Russello

Li-Pin Chang
Paola Lecca
Gwen Salaün

Seong-Je Cho
Byungjeong Lee
Patrizia Scandurra

Soon Ae Chun
Hong Va Leong
Jean-Marc Seigneur

Marília Curado
Paul Levi
Dongwan Shin

Egidio Falotico
Frédéric Louergue
Eunjee Song

Mario M. Freire
Sergio Maffiès
Junping Sun

João Gama
Cristian Mateos
Sangsoo Sung

Marisol García-Valls
Hernán Melgratti
Dan Tulpan

Marc-Oliver Gewaltig
Arianna Meniassi
Stefan Ulbrich

Raúl Giráldez
Mercedes G. Merayo
Julita Vassileva

Karl M. Goeschka
Marjan Mernik
Teresa Vazão

Aniruddha Gokhale
Riichiro Mizoguchi
Hugo Torres Vieira

George Hamer
Marco Di Natale
Tomas Vojnar

Hyoil Han
Rui Oliveira
Wei Wang

Ramzi A. Haraty
Ganesh Kumar P.
Raymond Wong

M. Ani Hsieh
Apostolos N. Papadopoulos
Davide Zambrano

Jun Huang
Gabriella Pasi


APPLIED COMPUTING REVIEW DEC. 2016, VOL. 16, NO. 4

3
SIGAPP FY’16 Quarterly Report

October 2016 – December 2016
Jiman Hong

Mission

To further the interests of the computing professionals engaged in the development of new computing applications and to transfer the capabilities of computing technology to new problem domains.

Officers

Chair
Jiman Hong
Soongsil University, South Korea

Vice Chair
Tei-Wei Kuo
National Taiwan University, Taiwan

Secretary
Maria Lencastre
University of Pernambuco, Brazil

Treasurer
John Kim
Utica College, USA

Webmaster
Hisham Haddad
Kennesaw State University, USA

Program Coordinator
Irene Frawley
ACM HQ, USA

Notice to Contributing Authors

By submitting your article for distribution in this Special Interest Group publication, you hereby grant to ACM the following non-exclusive, perpetual, worldwide rights:

- to publish in print on condition of acceptance by the editor
- to digitize and post your article in the electronic version of this publication
- to include the article in the ACM Digital Library and in any Digital Library related services
- to allow users to make a personal copy of the article for noncommercial, educational or research purposes

However, as a contributing author, you retain copyright to your article and ACM will refer requests for republication directly to you.

Next Issue

The planned release for the next issue of ACR is March 2017.
Automated Extraction of Business Documentation in Enterprise Information Systems

Karel Cemus
Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic
cemuskar@fel.cvut.cz

Tomas Cerny
Faculty of Electrical Engineering
Czech Technical University in Prague
Prague, Czech Republic
tomas.cerny@fel.cvut.cz

ABSTRACT

Extracting business documentation from ever-evolving information systems is a challenging task that requires a lot of effort, focus and technical expertise. Such a documentation provides a detailed overview of the system, lists domain model, operations, pre and post-conditions that impact the business flow. While all this information is captured by a particular software system, domain experts interested in such knowledge have limited access to it, either due to the lack of technical and programming expertise, or because the knowledge tangles throughout the entire system.

This paper introduces a novel approach in aspect-driven information systems enabling automated extraction of business documentation and its transformation to various perspectives allowing domain experts to acquire needed and up-to-date information and knowledge. Moreover, our approach makes it possible to transform system business rules into a formal language description verifiable by checkers validating the feasibility of business operations. We demonstrate the approach in a case study indicating its benefits.

CCS Concepts

- Software and its engineering → Domain specific languages;
- Software development methods;
- Layered systems; System description languages; Software design engineering; Formal software verification;

Keywords

Enterprise Information Systems; Aspect-Oriented Programming; Model-Driven Development; Business Documentation.

1. INTRODUCTION

Enterprise Information Systems (EISs) belong among web applications maintaining large volume of data with respect to the given business domain. This domain defines business processes consisting of steps and business operations. Each of these operations is constrained by a set of preconditions and post-conditions determining its role within the process.

The conventional design approaches [8, 20] often suggest the three-layered architecture [11], which divides the application into three layers: 1) the Persistence layer accessing the persistent storage, 2) the Application layer implementing business processes, and 3) the Presentation layer exposing API and user interface. Unfortunately, business rules cross-cut throughout the whole system [5]. As is shown in Figure 1, within a single request execution, the rules are applied in:

- \( \text{Pr} \) for first client-side, then server-side input validation,
- \( \text{A} \) to verify preconditions of a business operation,
- \( \text{Pe} \) to protect integrity constraints of the data storage,
- \( \text{A} \) to apply post-conditions to filter output data,
- \( \text{Pr} \) to restrict access to protected components.

Such distribution easily introduces inconsistencies among the captured business rules. Keeping all places synchronized is very challenging, especially, when contemporary development approaches suggest agile life-cycle. It is usually iterative and incremental [13, 21], i.e., during evolution the system changes many times. For example, we evolve model classes and their relations, implemented business operations and their constraints, as well as user interface. Furthermore, during such development, documentation of business processes gets often obsolete, which complicates resolution of identified inconsistencies.

Unfortunately, contemporary design and development approaches do not provide any efficient mechanism to review implemented business processes nor constraints of business operations. It usually results from the kind of their representation in the system. Contemporary approaches tangle business rules into source code, often written in multiple programming languages [5]. For example, we use Java for \( \text{Pe} \) and \( \text{A} \) implementation, SQL for database querying in \( \text{Pe} \), and HTML with JavaScript for UI implementation in \( \text{Pr} \). That makes the rules extraction difficult. In consequence, we either must review the implemented rules manually, or fall back to complex and inaccurate reverse engineering.

In this paper, we propose automated business documentation derivation technique, which extracts the business opera-

\[1\] By business rules we understand all preconditions and post-conditions of processes and their operations identified in the business domain as well as all model constraints [6].
You are a helpful assistant. Do not hallucinate.
Business rule-oriented conceptual modeling is a special case. It works with Event-Condition-Action (ECA) rules [3, 19] and utilize tailored Domain-Specific Languages (DLSs), e.g., JBoss Drools. These languages are efficient in expressing business rules, but in expressing actions they often behave like general-purpose languages. However, these usually easily readable and self-documenting languages still tangle business rules throughout multiple actions, as they do not consider rules separation, encapsulation and reuse. Furthermore, these systems optimize rules evaluation in expense of inspection, which makes them difficult to transform and reuse. In consequence, business rules cannot be neither easily extracted for documentation purposes nor transformed and reused in within the system.

2.3 Business Rules Extraction

When the project documentation is missing, or gets obsolete and used technologies does not provide mechanism for business rules extraction, we must reverse engineer the code to identify the processes and their rules. There exist several approaches and frameworks to business rules reverse engineering. For example, we may apply phrasal pattern matching on source code to extract the rules [24] or construct a call-graph and look for branching [26]. Either way, we must identify the execution context, i.e., variables and their origin used in extracted expressions. Generation of such documentation is challenging and the result may be inaccurate depending on the technology and code conventions.

Importance but difficulty of business rules extraction from legacy information systems is discussed in [25]. The authors state that while contemporary design approaches require repetition of business rules, all places are synchronized in the beginning. Later, during the system evolution, the overview of the system gets lost and inconsistencies are introduced. Subsequently, after several maintenance cycles, nobody knows what the system actually implements. Then, extract the business rules and processes from the system is nearly impossible, since there is no way to determine the truth. [25] proposes a semi-automated technique to extract the rules, but as it is obvious, such a documentation would require significant efforts, be very inaccurate, and could not be up to date.

Difficulty of business rules encapsulation and subsequent automated extraction lies in their characteristic. As they are considered throughout the whole system, they cross-cut multiple layers, components, and often technologies. Unfortunately, commonly used Object-oriented programming fails in encapsulation of such cross-cutting behavior [17], and tend to their manual tangling and duplication in a code base. Their separation is very difficult [15] due to the necessity to apply them in various places and technologies [5].

Despite the importance of business rules, their cross-cutting behavior makes their extraction from existing EISs challenging [9]. The authors proposed heuristics extending Model-Driven Engineering with BREX extraction method to extract business rules from existing Java applications. They use graph techniques with text analysis to parse captured business rules. Unfortunately, although they acknowledge business rules repetition and try to trace business rules to their source, they do not discuss how they deal with colli-

2. ASPECT-DRIVEN DESIGN APPROACH

Alternative ADDA concept extends common three-layered architecture [8, 20] and focuses on encapsulation and automated reuse of cross-cutting concerns including business rules [5]. The authors consider the business rules to be a cross-cutting concern and utilize Aspect-Oriented Programming (AOP) [17] to encapsulate them as an aspect in a single focal point. Such encapsulation significantly simplifies both development and maintenance, and reduces error-proneness as every rule is represented only once. Common Object-Oriented approach would result in extensive repetition of rules in source code.

ADDA groups the rules up into business contexts, which encapsulates all preconditions and post-conditions belonging to a single business operation. Furthermore, likewise both considered reverse engineering methods, also ADDA considers the rules within an execution context. For each evaluation, it binds the variables considered within the business context to concrete values. Such a mechanism enables the context-awareness while it preserves its simplicity.

In scope of this paper, the basic idea of ADDA lies in considering both the business rules and business objects to be cross-cutting concerns. Then, in terms of AOP, it encapsulates business rules grouped into business contexts in an aspect implemented as the business rules registry, which we show in Figure 2. In this registry, the rules are described
4. BUSINESS DOCUMENTATION

Business documentation presents the overview of the captured business, i.e., the logic, objects, and all business rules. In conventional approaches, we have to either dedicate some resources to documentation maintenance in our software development model or use some of reverse engineering techniques to acquire current documentation of the EIS.

4.1 Business Documentation with ADDA

Systems following ADDA concept describe constraints of business operations in DSL and group the business rules up into business contexts around business operations. This encapsulation is very convenient for inspection, e.g., for documentation purposes. This representation makes easy answering questions about the rules of the particular operations. In consequence, we reduce the problem of business rules extraction to detection of business operations.

Contemporary EIS design including ADDA often follows the three-layered architecture [8, 11, 20]. It decomposes the system into the three layers, where the middle Application layer implements the business logic. Furthermore, this architecture follows the Transaction script design pattern [11], which results in business operation-centric design of this layer. In consequence, all so called services are highly cohesive containers of public API exposing business operations. To extract the operations, we just need to read API of (A).

4.2 Documentation Derivation Process

Process of business documentation derivation in ADDA systems consists of several steps, as we show in Figure 3. Here, we elaborate each step to discuss the flow.

4.2.1 Identification of Services.

First, we identify all services. In terms of the three-layered architecture, these are objects encapsulating the business logic in (A). They are procedural containers following Transaction script design pattern. Example of such a service is shown in Listing 2. The intention of the pattern lies in design of highly cohesive objects encapsulating all logic related to a specific, in this case business, object/component. This makes the business documentation extraction much easier. Especially, when all contemporary technologies following this architecture use Inversion of Control pattern (IoC) with Dependency Injection [11]. In such an environment, all services are marked with meta-instructions to instruct the IoC container how to handle the class. In consequence, to locate all services, we just scan all classes for the proper markup sign, e.g., an annotation like in Listing 2. Output of this step is the list of objects encapsulating all business operations related to a business object/component.

---

Listing 1: Business context: Report an Issue

```
rule "Pre: Report Issue" when
   Issue(
      # non-empty description, but min length
      title != null && title.length > 10,
      # max length
      title != null || title.length < 200,
      # pattern check on title
      title matches "^[a-zA-Z0-9 ]*$",
      # max length with long texts
      description != null || description.length < 1000,
      # min max value
      priority >= 1 && priority <= 3,
      # required
      type != null
   )
   $current user must be authenticated
   eval( $user.isAuthenticatted )
end
```

---

in some convenient, inspectable and usually domain-specific language to enable their further transformation and automated reuse.

Consider the example in Listing 1. We have an issue tracking system with Report an Issue operation. The example shows the powerful but general business context declaration in JBoss Drools DSL. The Issue model class has several constraints on its fields and the current user have to be authenticated. Unfortunately, this language is a bit inefficient for this use, which requires the Pre prefix as we are unable to group both preconditions and post-conditions [5].

To avoid manual code repetition and prevent inconsistencies in source code, it inspects the rules and automatically transforms them into various components, layers, and technologies [4]. This transformation is enabled by the use of DSL for the rules description and powerful aspect weavers producing the resulting context-aware code at runtime. We activate it by addressing the context to apply. However, the aspect weavers are quite complex and specific to every tuple of output component and technology, which represents major overhead. Fortunately, they are reusable among projects.

Using ADDA in EIS development, we benefit from having a single focal point declaring all cross-cutting concerns such as business rules, UI layouts, and widgets. ADDA implementation takes responsibility of automated distribution and restatement of all those concerns, which significantly reduces development and maintenance efforts and error-proneness, while it eases testability. However, use of multiple DSLs increases initial overhead and mental complexity. In addition, it requires implementation of multiple aspects weavers accepting all used DSLs and producing target components in the specific technologies [4, 5].
4.2.2 Identification of Business Operations.

Each service consists of a set of public methods exposed into the application layer. Using Transaction script pattern, all these methods in the single service correlate to the business operations related to the single business object/component, based on the service design. In consequence, inspecting public methods of services, we extract business operations.

Each business process and subsequently each business operation are restricted by set of business rules, i.e., preconditions and post-conditions. Those must be always satisfied within current execution context, i.e., bound contextual variables such as user’s identity, and the application state. ADDA encapsulates these rules in business contexts mapped to business operations. Therefore, inspecting the context related to the particular operation we acquire the list of its business rules. As Listing 2 shows, with ADDA, each business operation must address its business context, e.g., in an annotation. Then, we are able to read it and apply when necessary. In this case, we use the contained information to produce business documentation.

4.2.3 Identification of Business Objects.

Besides the operations restricted by preconditions and post-conditions, there are business objects and their attributes both possibly constrained by integrity constraints, i.e., invariants. These objects are used in X and Y to carry data and often overlap with the domain/data model. In ADDA terminology, their invariants are considered among business rules, and definitely must be published in business documentation along with the structure of the objects.

Contemporary EISs following three-layered architecture use Object-Relational Mapping frameworks such as Java Persistence API. These use meta-instructions such as annotations to identify domain model classes. As business objects often overlap with the model to simplify the implementation, we may use their markup annotations such as @Entity to identify them. Then, together with related business context declaring their invariants, we inspect them to analyze their structure and relations, also marked with annotations such as @ManyToMany.

Finally, having mined all information, we proceed to actual business documentation generation, i.e., organization of the information into views with some business value.

4.3 Views on Business Documentation

We mine various information such as 1) objects encapsulating business operations, 2) business operations, their business rules and execution contexts, and 3) business objects, their structure, relations and invariants. All together is major volume of information, and each stakeholder is interested in a bit different subset of them. To improve the documentation delivery, we are able to produce different views on gathered information to fit their needs.

4.3.1 List of Business Operations.

First, it is common requirement to list all supported use-cases/scenarios to get overview of the current state of the system. Having a list of business operations, we are able to provide simple list of implemented use-cases grouped by the business object, as is shown in Listing 3. Project managers as well as customers might be interested in this view.

4.3.2 Validation of Business Contexts.

Contemporary systems often suffer from inconsistencies in captured business rules as they are repeated many times...
in the code base. ADDA focuses on this issue with the rules encapsulation in the single focal point and then automated transformation and distribution into the whole system. However, the rules still must be validated by domain experts to ensure the system behaves correctly. Even though ADDA uses DSL to reduce the initial mental barrier for domain experts, sometimes it is convenient to list all operations and their contexts to let the experts review them. Listing 4 shows an example of such a detailed view on a single operation. The output is just inspected and transformed DSL, which makes it convenient to read.

### 4.3.3 Domain Model Structure.

One of the crucial views is the domain model structure, i.e., identification of business objects, their attributes, and relations. Due to Object-Relational Mapping in use and we can simply inspect classes and fields for proper meta-instructions and easily construct the view, possibly in UML. We are also able to determine relations multiplicity and add integrity constraints transformed from the business context.

### 4.3.4 Business Contexts Feasibility.

Finally, EISs are often tested but not verified, as it is very difficult to extract formal system specification. Inspecting business rules DSL, we are able to transform the rules representation into any formal specification and perform some reasoning over it. For example, we may verify the rules against a formal specification, automatically determine the feasibility of the contexts, i.e., whether there exist values satisfying the assumptions, and find cycles among contexts as they may include each other. This view may significantly help to automation of system testing and error detection.

### 5. CASE STUDY

To demonstrate the business documentation derivation technique, we implement an issue tracking system with a domain model shown in Figure 4. The system maintains multiple projects, where each project has developers assigned to issues. The system identifies three user roles with corresponding scenarios: 1) regular users report issues, 2) developers are assigned to and resolve issues, and 3) administrators maintain both projects and users.

We implement this system in the three-layered architecture using Java EE 7, with JBoss Drools used for business rules representation. We use the aspect weavers proposed in [4, 5, 6], and extend them with the capability to produce documentation basing on the details provided in previous Section. Running the documentation generators on this system produces several possible outputs.

### 5.1 Properties of Case Study

Business documentation consists of several aspects and has multiple points of view. We focused our case study on some of them, which we elaborate in the following sections.

#### 5.1.1 List of Business Operations.

For each service, i.e., class annotated with @Service in 

Listing 4. For the single operation, it shows the list of expected parameters to the business operation, e.g., single issue. We render this overview in computer-friendly XML format, see Listing 5. Then, we are able to apply transformation rules such as XSLT to produce, e.g., user-friendly HTML with clickable links, a checklist reviewed by domain experts, and fill in scenario templates proposed in [20].

#### 5.1.2 Validation of Business Contexts.

Business experts are interested in the rules applied on the business operations. Inspecting all services, we are able to identify the business operations, their contexts, and the rules they consist of. Then, we can list and possibly transform (e.g., group up) them. Example of the output we show in Listing 4. For the single operation, it shows the list of expected parameters to the business operation, e.g., single issue object, and the list of contextual parameters retrieved from any available context, e.g., a current User instance. Next, it prints security assumptions, and finally both preconditions and post-conditions. Furthermore, we also deliver this overview in computer-friendly XML format, see Listing 5. Then, we are able to apply transformation rules such as XSLT to produce, e.g., user-friendly HTML with clickable links, a checklist reviewed by domain experts, and fill in scenario templates proposed in [20].

Listing 5: Details of business operation in XML

```xml
<context class="IssueService" operation="report">
  <contextual-var name="user" type="User"/>
  <business-var name="issue" type="Issue"/>
  <security>user.isAuthenticade</security>
  <pre attribute="title">is required</pre>
  <pre attribute="title">has <10,200> chars</pre>
  <pre attribute="title">matches "^[a-zA-Z0-9 \-]+$"</pre>
  <pre attribute="description">is optional</pre>
  <pre attribute="priority">is from <1,3></pre>
  <pre attribute="priority">is shorter than 1000</pre>
  <pre attribute="priority">is required</pre>
</context>
```

Figure 4: Model of the issue tracking system

Having detailed overview of each business operation, domain experts are able to quickly validate it, compare to the specification, and help to identify possible errors as soon as possible without any programming knowledge.

5.1.3 Domain Model Structure.

Domain model structure retrieval is quite easy task already published in many papers (see related work for more details). Using JPA with @Entity marking domain model classes, and @OneToMany and @ManyToOne identifying the type of the relation, significantly simplifies the task. However, the model structure also belongs into business documentation to deliver overall overview of the system. We produce structured textual overview such as in Listing 6, and an XML document. Then, we can transform the document and visualize it in an existing UML visualizer. However, implementing the transformation might be difficult as UML visualizers often work with complex descriptions. At the same time [16] shows that various formats are interoperable in various CASE tools.

5.1.4 Business Contexts Feasibility.

Finally, an interesting use of this documentation is the formal verification of business context feasibility. To demonstrate the possibility of the DSL transformation into a formal language, we transformed the rules into OCL. The example of the output is shown in Listing 7. It describes the same context as we show in Listings 4 and 5 but in OCL language. Possibly, we could implement transformation to any other language and then run the specification through some formal specification checker.

6. CONCLUSION

Derivation of business documentation from an existing EIS is very challenging. Identification of all business operations and then subsequent extraction of their preconditions, post-conditions and security constraints is not trivial task. There are several challenges including missing single point of truth, business rules being tangled throughout multiple layers, components, and technologies, and resolution of inconsistencies introduced in maintenance as a result of their repetition.

In this paper, we introduce a novel technique of documentation derivation from EISs following ADDA concept. These systems isolate business rules in a specialized registry, and then automatically transform and propagate them throughout the whole system at runtime. We implement aspect weavers inspecting the business rules represented in DSL, and identifying services and business operations in the code to produce various business views on the system.

These descriptions are produced in either human-readable textual or well-structured XML format to simplify further processing. XML documents are easily transformable using XSLT into different presentations, e.g., interactive HTML, a simple checklist, and scenario templates. Structured XML description of the domain model architecture is visualizable in an existing UML visualizers. Having these current, reliable, and easily generated reports significantly simplifies development process. It is easier to involve domain experts to collaborate on business rules description and to review the current state. Finally, internal business rules representation is transformable into formal languages such as OCL, which enables formal verification of business context feasibility.

While many reverse engineering-based techniques browse the existing source code, parse it, and perform graph operations to extract business rules, this technique builds on ADDA concept decomposing cross-cutting concerns in the system. In consequence, it requires the system to be designed with ADDA since the beginning, the technique cannot be applied on existing legacy systems. Unfortunately, the setup of the ADDA concept introduces significant initial overhead and requirement of advanced implementation of aspect weavers. However, it may be continuously deployed into existing systems to ease the transformation. In future work, we will focus on use of the formal specification of the current system to conduct valuable verifications.

Acknowledgments

This research was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS14/198/OHK3/3T/13.

7. REFERENCES


Listing 6: Domain Model Structure

| Class: Issue |
| Attributes: id:Long, description:String, log:String |
| Relations: belongsTo:Project(1:N), assignee:User(0..1:N), reporter:User(1:N) |

| Class: Project |
| Attributes: id:Long, name:String |
| Relations: issues:Issue(N:1), developers:User(N:M) |

| Class: User |
| Attributes: id:Long, firstName:String, lastName:String, role:Role |
| Relations: assignments:Issue(N:1), reported:Issue(N:1) |

Listing 7: Business context in OCL language

```ocl
class Issue
  attributes
    id: Long,
    description: String,
    log: String

  relations
    belongsTo: Project(1:N), assignee: User(0..1:N), reporter: User(1:N)
end class

class Project
  attributes
    id: Long,
    name: String

  relations
    issues: Issue(N:1), developers: User(N:M)
end class

class User
  attributes
    id: Long,
    firstName: String,
    lastName: String,
    role: Role

  relations
    assignments: Issue(N:1), reported: Issue(N:1)
end class
```

context

```
  context Issue::report(user: User)
  pre:
    user.isAuthenticated = true and
    title <> null and
    title.size() >= 10 and
    title.size() <= 200 and
    title.regexMatch("^[a-zA-Z0-9 ]*$") and
    (description = null or description.size() <= 1000) and
    priority >= 1 and priority <= 3 and
    type <> null

  post:
    true
```
ABOUT THE AUTHORS:

Karel Cemus received his Bachelor’s and Master’s degrees from the Faculty of Electrical Engineering of Czech Technical University in Prague, Czech Republic. Since 2013, he is a Ph.D. candidate in Information Science and Computer Engineering at this university. His area of research is software engineering, separation of concerns, model-driven development, and enterprise application development.

Tomas Cerny received his Bachelor’s and Engineer’s degrees from the Faculty of Electrical Engineering of Czech Technical University (FEE, CTU) in Prague, Czech Republic. Next, he received his M.S. degree from Baylor University and in 2016 finished Ph.D. in Information Science and Computer Engineering. Since 2009, he works as an Assistant Professor at FEE, CTU. His area of research is software engineering, separation of concerns, model-driven development, enterprise application development and networking.
Scheduling Algorithm for Parallel Real-Time Tasks on Multiprocessor Systems

Chin-Fu Kuo  
Department of Computer Science  
and Information Engineering  
National University of Kaohsiung  
Kaohsiung, Taiwan, R.O.C.

Yung-Feng Lu  
Department of Computer Science  
and Information Engineering  
National Taichung University of Science and Technology  
Taichung, Taiwan, R.O.C.  
yflu@nutc.edu.tw

ABSTRACT
The purpose of this paper is to study the task scheduling problem of task sets on multiprocessor systems. In the task sets there are parallel tasks and sequential tasks. Parallel tasks can not meet their deadlines if they are executed by one unique thread. However, a parallel task has several parallelization options. A good parallelization level for a parallel task can make it meet its deadline and result in the addition of extra execution time due to parallelization overhead. We propose the Best-Fit based on Equal Slack (BEES) algorithm for deadline setting and task assignment. To derive a feasible task assignment, we must select a proper parallelization level from the available parallelization options for each parallel task. Then each parallel task will be split into several subtasks. Finally, sequential tasks and generated subtasks for parallel tasks are assigned to processors. A series of experiments were conducted to evaluate the proposed algorithm. From the experimental results, we can observe that the proposed algorithm had better performance than the compared algorithms. The experimental results demonstrate that the performance of the algorithms using the Equal Slack strategy is better than that using the Equal Flexibility strategy.

CCS Concepts
• Computer systems organization → Embedded systems; Redundancy; Robotics; • Networks → Network reliability;

Keywords
Deadline assignment; Task assignment; Multiprocessor systems; Parallelization overhead; Parallelization option

1. INTRODUCTION
Since processor frequency is bounded due to physics constraints like power dissipation, multi-core architectures have become the solution to allow performance to keep growing as described by Moore’s law. These architectures present an interesting challenge: produce applications which fully exploit the parallelism provided by these processors. Recently, most modern real-time applications are becoming computation-intensive and need more computation power. Approaching the theoretical performance of these architectures is a complex issue[4, 8]. An application can not meet its timing constraints if it is executed by a thread. Hence, the applications are parallelized into multiple CPU threads, and the high-performance computation power from multiple cores on the platforms are used to execute them[1, 16, 18]. Some popular parallel programming models, such as OpenMP[5] and OpenCL[14], are applied to develop parallel programming applications.

OpenMP and OpenCL provide the fork-join task model to extend to the parallel execution from the original sequential execution of an application. A fork-join application consists of sequential segments which only can be executed by a single thread and parallel segments which can be executed by multiple threads. The application started by executing sequentially and then forks to be executed in parallel. After the parallel execution is completed, the results from different threads are aggregated by performing an implicit/explicit join operation. The number of threads to execute a parallel segment can be controlled by the program developers. Therefore, the execution time to complete the segment significantly varies depending on the thread number. For the execution of each parallel segment, one of multiple parallelization options can be selected. However, the parallelization can add extra execution time due to the synchronization and data communication among the threads, called parallelization overheads. Therefore, the total execution time for a parallel segment at a higher parallelization level, i.e., more threads, can be significantly larger than that at a lower parallelization level.

If a segment of code can not be parallelized and only be able to executed by one thread, the segment is called a sequential segment. A segment of code without dependencies can be executed by more than one thread in parallel and is...
called a parallel segment. OpenMP [5] is a shared-memory application programming interface (API). The directives of OpenMP can be added to a sequential program in Fortran, C, or C++ to describe how the work is to be shared among threads that will execute on different processors. The directives let a programmer tell the compiler which instructions to execute in parallel and how to distribute them among the threads that will run the code statements. An OpenMP directive is an instruction in a special format that is understood by OpenMP compilers only, e.g., Microsoft Visual Studio C++ and the GNU Compiler Collection (gcc)[15]. OpenMP is applied to incrementally create a parallel program from an existing sequential code. A programmer can reorganize portions of a code to obtain independent instruction sequences in a program and then insert directives into the portions. The rest of the program is left in its sequential form. In the OpenMP programming model, the execution of a parallel program is started by an initial single thread which is the master thread of the program. When the master thread encounters a parallel construct with parallel pragma (i.e., the starting point of the parallel execution), a team of threads is created to execute the instructions in the parallel pragma in a parallel manner. The clause num_threads(.) can be used to set the number of threads in a thread team including the major thread. The procedure is called the fork-join programming model.

The targeted scheduling problem is that there are sequential tasks and parallel tasks in the system with multiprocessor. The parallel tasks are implemented with OpenMP[5] or OpenCL[14]. The thread number for a parallel task can be set before the system starts to execute. We want to propose an off-line assignment mechanism to assign tasks to processors. The problem of task assignment for a multiprocessor system is NP-Hard [3, 7, 9, 13]. Several heuristics, including the First-Fit, Best-Fit, Next-Fit, and Worst-Fit algorithms, were proposed. The Worst-Fit (WF) heuristic allocates a task to the processor with the highest enough available capacity. The Best-Fit (BF) heuristic allocates a task to the processor with the lowest enough available capacity. The First-Fit (FF) heuristic allocates a task to the first processor with enough available capacity. Such heuristics can not be directly applied to the scheduling problem. A parallel task needs computation power from more than one processor. If the task is executed on a processor, it will miss its deadline. In other words, the total execution time required from one processor is larger than its period, i.e., the utilization larger than 1.0. To meet its deadline, the task must be executed simultaneously by multiple processors. We propose the Best-Fit based on Equal Slack (BFES) algorithm to solve the scheduling problem. The parallel segment of a parallel task can be parallelized into multiple independent threads. The thread number is called parallelization level. The parallel segment has M parallelization options, where M is the processor number. In order to meet the deadline of a parallel task, its parallel segment must be executed by more than one thread. First, we adopt the deadline assignment strategy, Equal Slack (EQS) [6, 11], to translate the precedence constraints among the subtasks of a parallel task into the timing constraints. We determine the thread number, i.e., the number of subtask, for the parallel segment of the parallel task. Each subtask is assigned a relative deadline. If all the subtasks meet their deadlines, the overall task meets its end-to-end deadline. Then, the Best-Fit heuristic is used to assign each sub(task) to a proper processor. Kwon et al. proposed an optimal algorithm for parallelizing and scheduling a task set with multiple parallelization options on multiple processor systems [10]. The algorithm presented in [10] is a global strategy while our proposed algorithm is a partitioning strategy.

The rest of this paper is organized as follows: Section 2 defines the system model and formalizes the target problem under considerations. Section 3 presents a motivational example and the Best-Fit based on Equal Slack (BFES) algorithm for scheduling problem where the task set execute on a multiprocessor platform. Simulation experiments are presented in Section 4 to evaluate the capacity of the proposed algorithm. Section 5 is the conclusion.

2. SYSTEM MODEL

In the paper, we consider the task scheduling problem where a set $T = \{\tau_1, \tau_2, ..., \tau_N\}$ of periodic real-time tasks is executed on a multiprocessor platform equipped with a set $M = \{P_{r1}, P_{r2}, ..., P_{rM}\}$ of M identical processors. We assume that the tasks in the task set T are independent from each other.

In $T$, there are a sequential task subset $T_s$ and a parallel task subset $T_p$. A task with one parallel segment possibly executed by more than one thread, is regarded as a parallel task while a task only with a sequential segment is called a sequential task. A parallel task starts by a master thread executing sequentially and then forks to be executed in parallel on more than one processors. When the parallel execution has completed on each of the used processors, the results are aggregated by performing a join operation and the execution of the sequential segment is resumed within the master thread. Therefore, we assume that each instance of a parallel task consists of three segments. The first segment and third segment are sequential ones which only can be executed by a thread while the second segment is a parallel one which can be executed by more than one threads.

Each task $\tau_i$ in the sequential task subset $T_s$ is defined by a 3-tuple $\{T_i, C_i, D_i\}$, where $T_i$, $C_i$, and $D_i$ are its period, worst-case execution time, and relative deadline, respectively. Besides, $D_i$ is equal to $T_i$. The utilization $u_i$ of task $\tau_i$ is defined as $\frac{D_i}{T_i}$. Each task $\tau_i$ in the parallel task subset $T_p$ is defined by a 3-tuple $\{T_i, C_i, D_i\}$, where $T_i$ and $D_i$ are its period and relative deadline, respectively. $D_i$ is equal to $T_i$. However, $C_i$ is the vector of the worst-case execution times of for the three segments, i.e., $c_{i,1}$, $c_{i,2}$, and $c_{i,3}$. $c_{i,1}$ and $c_{i,3}$ are the worst-case execution times of the first segment $s_{i,1}$ and the third segment $s_{i,3}$ of task $\tau_i$, and $s_{i,1}$ and $s_{i,3}$ are only executed sequentially. $c_{i,2}$ is the vector of the worst-case execution time of the second segment $s_{i,2}$ executed at different parallelization levels, i.e., different number of processor threads. In other words, the second segment $s_{i,2}$ can be parallelized into $x$ threads, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$. We say that there are $M$ parallelization options for $s_{i,2}$. $c_{i,2,x}$ is defined as the worst-case execution time of the second segment $s_{i,2}$ executed at level $x$, where $1 \leq x \leq M$.
time needed by each of \( x \) processor threads for the second segment \( s_{i,2} \). To meet the deadline a parallel task needs more computing power from multiple processors. It can not meet its deadline if it is executed just by a processor. In other words, \( (c_{i,1} + c_{i,2,1} + c_{i,3}) \geq T_i \). To meet its deadline \( \tau_i \) must be executed by more than one thread. We assume that the second segment \( s_{i,2} \) can be evenly distributed among the used threads. The precedence constraints among the three segments must be maintained. The execution for the second segment \( s_{i,2} \) can not start until that of the first segment \( s_{i,1} \) is completed. On the other hand, the execution for the third segment \( s_{i,3} \) can be started after all of the threads for the second segment \( s_{i,2} \) finish their executions. Figure 1 gives a clear illustration. Let define \( d_{i,1,x}, d_{i,2,x}, \) and \( d_{i,3,x} \) as the relative deadlines for \( s_{i,1}, s_{i,2}, \) and \( s_{i,3} \) when \( s_{i,2} \) is executed by \( x \) threads. Note that \( T_i = d_{i,1,x} + d_{i,2,x} + d_{i,3,x} \). We denote \( \delta_{i,2,x} \) as the density of one of the \( x \) threads for the second segment \( s_{i,2} \) for a parallel task \( \tau_i \) if \( s_{i,2} \) is executed by \( x \) processor threads. We also define \( \delta_{i,1,x}/(\delta_{i,3,x}) \) as the density of the first/(third) segment for \( \tau_i \). The density \( \delta_{i,1,x}/(\delta_{i,3,x}) \) is equal to \( \frac{c_{i,1}}{d_{i,1,x}/d_{i,3,x}} \). The second segment \( s_{i,2} \) is executed by \( x \) threads simultaneously and then the density \( \delta_{i,2,x} \) of each thread is equal to \( \frac{c_{i,2,x}}{d_{i,2,x}} \). Assume that \( \tau_i \) is released at time \( t \). The execution of the segment \( s_{i,1} \) must be started from \( t \) to its absolute deadline \( t + d_{i,1,x} \). The execution of the segment \( s_{i,j} \), where \( j \geq 2 \), must be in the interval \( t + d_{i,1,x} + \ldots + d_{i,j-1,x} \) to its absolute deadline, i.e., \( t + d_{i,1,x} + \ldots + d_{i,j-1,x} + d_{i,j,x} \). We call the interval as the active interval. In the next section, we will discuss how to set the relative deadlines \( d_{i,1,x}, d_{i,2,x}, \) and \( d_{i,3,x} \) for the three segments. Note that there are \( x \) threads for the second segment \( s_{i,2} \) and the total density of the \( x \) threads is equal to \( \frac{c_{i,2,x}}{d_{i,2,x}} \). \( c_{i,2,1} \) is much larger than \( c_{i,1} \) and \( c_{i,3} \). Considering the parallelization overheads, for the segment \( s_{i,2} \) of a parallel task \( \tau_i \) the total worst-case execution time under a larger number \( x \) of threads is larger than that under a smaller number \( y \) of threads, i.e., \( c_{i,2,x} \times x > c_{i,2,y} \times y \) if \( x > y \). For the first or third segment, there is the corresponding subtask to execute. However, the subtask number for the second segment is equal to the parallelism level, i.e., the used thread number. The utilization \( u_i \) of task \( \tau_i \) executed without parallelism is defined as \( \frac{c_{i,1} + c_{i,2,1} + c_{i,3}}{T_i} \). Some real-time applications need powerful computing power. However, due to the computation limitation on a uniprocessor system, the timing requirements can not be achieved. With the computing resources on the multiprocessor system, the time required for processing can be reduced. Therefore, the timing constraints can be satisfied on the multiprocessor system. In this paper, we will focus on such a system where for each parallel task \( \tau_i, u_i > 1 \), and if the second segment is executed only on a processor, the real-time requirements can not be satisfied. We define \( U \) as the total utilization of the periodic task set \( T \), that is to say, \( U = \sum_{i=1}^{N} u_i \).

![Figure 1: A parallel task example](image)

A (sub)task-to-processor assignment \( A \) is denoted by \( M \) disjoint subsets \( \{A_1, \ldots, A_M\} \). A subset \( A_k \) which can include sequential tasks and the subtasks of parallel tasks is assigned to one corresponding processor \( P_{Tk} \) and is scheduled with the earliest deadline first (EDF) algorithm [12]. Let define the total density \( \Delta_k \) as the summing value of the total utilizations of the sequential tasks and the total density of the subtasks of parallel tasks assigned to processor \( P_{Tk} \). Therefore, a (sub)task-to-processor assignment is feasible if the following conditions are satisfied: \( \Delta_k \leq 1 \), for \( 1 \leq k \leq M \). In other words, the tasks and subtasks in \( A_k \) will be scheduled and executed on processor \( P_{Tk} \) without missing their deadlines if the total density \( \Delta_k \) is no more than 1. Finally, the sequential and parallel task can meet their deadlines.

**Problem Formulation**: Given a set \( T \) of periodic sequential tasks and parallel tasks, and a set \( M \) of identical multiple processors, we want to propose an off-line methodology to derive the proper parallelism level for each parallel task and find a feasible (sub)task-to-processor assignment such that each task in the system do not miss its deadline. The methodology determines which processor each sequential task in the given task set should execute on. For each parallel task, the methodology not only determines the number of threads (i.e., parallelism level) for its second segment but also which processor each subtask should be executed on. Such a task scheduling problem for a multiprocessor system is NP-Hard [3, 17] and can be resolved only by using heuristic approaches.
3. ASSIGNMENT ALGORITHM FOR TASK SET INCLUDING PARALLEL TASKS

In this section, first an easy scheduling example is given to illustrate the motivation of the proposed algorithm. Then, we describe the proposed task assignment algorithm, called as the *Best-Fit based on Equal Slack* (BFES), in detail. The rationale of the functions in the proposed algorithm will be introduced.

### 3.1 Motivational Example

Table 1: The worst-case execution time of subtasks for the parallel task $\tau_1$.

<table>
<thead>
<tr>
<th>Period</th>
<th>WCET of the First Segment $s_{1,1}$</th>
<th>WCET of the Second Segment $s_{1,2}$</th>
<th>WCET of the Third Segment $s_{1,3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_1$</td>
<td>100</td>
<td>1</td>
<td>110</td>
</tr>
</tbody>
</table>

Consider a sample task set consisting of seven periodic tasks $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6,$ and $\tau_7$. The task set will execute on a platform with three processors, i.e., $P_{r1}, P_{r2},$ and $P_{r3}$. $\tau_1$ is a parallel task and consists of three segments $s_{1,1}, s_{1,2},$ and $s_{1,3}$. $s_{1,1}$ and $s_{1,3}$ are sequential segments. $s_{1,2}$ is a parallel segment for which there are three parallelization options, and its timing information is presented in Table 1. Its period is set as 100 and the worst-case execution times (WCET) of the first segment and third segment are equal to 1. The WCETs of the second segment under different thread numbers, 1, 2, and 3, respectively, are 110, 61, and 44. The other tasks are sequential tasks. Their timing constraints are presented in Table 2.

Table 2: The worst-case execution time of subtasks for the sequential tasks.

<table>
<thead>
<tr>
<th>Period</th>
<th>WCET</th>
<th>Period</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\tau_2$</td>
<td>100</td>
<td>50</td>
<td>$\tau_5$</td>
</tr>
<tr>
<td>$\tau_3$</td>
<td>100</td>
<td>16</td>
<td>$\tau_6$</td>
</tr>
<tr>
<td>$\tau_4$</td>
<td>100</td>
<td>16</td>
<td>$\tau_7$</td>
</tr>
</tbody>
</table>

First, we adopt deadline assignment strategies to assign relative deadlines to the subtasks of the parallel task $\tau_1$. Table 3 shows the relative deadlines derived with the Equal Flexibility (EQF) strategy [6, 11]. Let denote $\delta_{i,j,x}$ as the density of the subtask $\tau_{i,j}$ when we select the parallelism level $x$, i.e., $x$ threads, for the segment $s_{i,2}$. The densities of the subtasks of $\tau_1$ under the thread number $x$ are the same, i.e., $\delta_{1,1,x} = \delta_{1,2,x} = \delta_{1,3,x}$. The densities are equal to 1.12, 0.63, and 0.46, respectively, under the parallelism levels 1, 2, and 3. The total density of the subtasks of $\tau_1$ is denoted by $\delta_{1,x}$ when the parallelism level is $x$. Besides, let define $\Delta_x$ as the sum value of $\delta_{1,x}$ and the total utilization of sequential tasks, i.e.,

$$\Delta_x = \delta_{1,x} + \sum_{i=2}^{7} u_i$$

$$= (2 + x) \times \delta_{1,2,x} + \frac{50}{100} + 2 \times \frac{16}{100} + \frac{15}{100} + 2 \times \frac{10}{100}$$

Because the total densities of the subtasks are larger than the processor number under different thread numbers, i.e., 1, 2, and 3, there may exist no feasible assignment for the task set when the EQF strategy is used.

For a general case, we can use the following equation to derive the total density $\delta_{i,x}$ of subtasks for a task $\tau_i$, i.e.,

$$\delta_{i,x} = \sum_{j=1}^{3} \delta_{i,j,x} = \delta_{i,1,x} + x \times \delta_{i,2,x} + \delta_{i,3,x}$$

$$= \frac{c_{i,1}}{d_{i,1,x}} + x \times \frac{c_{i,2,x}}{d_{i,2,x}} + \frac{c_{i,3}}{d_{i,3,x}}$$

$$= \frac{c_{i,1}}{c_{i,1} + \frac{c_{i,2,x}}{c_{i,1} + c_{i,2,x} + c_{i,3}} (T_1 - c_{i,1} - c_{i,2,x} - c_{i,3}) +}$$

$$x \times \frac{c_{i,2,x}}{c_{i,1} + c_{i,2,x} + c_{i,3}} (T_1 - c_{i,1} - c_{i,2,x} - c_{i,3}) +$$

$$\frac{c_{i,3}}{c_{i,1} + c_{i,2,x} + c_{i,3}} (T_1 - c_{i,1} - c_{i,2,x} - c_{i,3})$$

Let define $f(x)$ as the function of the WCET $c_{i,2,x}$ of the segment $s_{i,2}$ when the parallelism level is equal to $x$. Then we can rewrite $\delta_{i,x}$ as follows:

$$\delta_{i,x} = \frac{c_{i,1}}{c_{i,1} + \frac{c_{i,2,x}}{c_{i,1} + f(x)} + c_{i,3}} (T_1 - c_{i,1} - f(x) - c_{i,3}) +$$

$$x \times \frac{f(x)}{c_{i,1} + f(x)} (T_1 - c_{i,1} - f(x) - c_{i,3}) +$$

$$\frac{c_{i,3}}{c_{i,1} + f(x) + c_{i,3}} (T_1 - c_{i,1} - f(x) - c_{i,3})$$

### Table 3: The relative deadline of each subtask under different thread numbers when the Equal Flexibility strategy is adopted.

<table>
<thead>
<tr>
<th>Thread Number for $s_{1,2}$</th>
<th>$d_{1,1}$</th>
<th>$d_{1,2}$</th>
<th>$d_{1,3}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.89</td>
<td>98.22</td>
<td>0.89</td>
</tr>
<tr>
<td>2</td>
<td>1.59</td>
<td>96.82</td>
<td>1.59</td>
</tr>
<tr>
<td>3</td>
<td>2.17</td>
<td>95.66</td>
<td>2.17</td>
</tr>
</tbody>
</table>
We assume that \( C_1 = c_{i,1} + c_{i,3} \) and \( C_2 = T_i - C_1 \). Then, the equation can be rewritten as follows:

\[
\delta_{i,x} = \frac{1}{1 + \frac{c_2 - f(x)}{C_1 + f(x)}} \times \frac{1}{1 + \frac{c_2 - f(x)}{C_1 + f(x)}} + \frac{1}{1 + \frac{c_2 - f(x)}{C_1 + f(x)}} = (2 + x)\frac{1}{1 + \frac{c_2 - f(x)}{C_1 + f(x)}}
\]

\[
= (2 + x) \frac{C_1 + f(x)}{C_1 + C_2}
\]

We assume that \( C = \frac{1}{1 + x^2} \). Then, the equation can be reformatted as follows:

\[
\delta_{i,x} = (2 + x) (C_1 + f(x))
\]

**Table 4:** The relative deadline of each subtask under different thread numbers when the Equal Slack strategy is adopted.

<table>
<thead>
<tr>
<th>Thread Number for ( s_{1,2} )</th>
<th>( d_{i,1} )</th>
<th>( d_{i,2} )</th>
<th>( d_{i,3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-3</td>
<td>106</td>
<td>-3</td>
</tr>
<tr>
<td>2</td>
<td>13.33</td>
<td>73.34</td>
<td>13.33</td>
</tr>
<tr>
<td>3</td>
<td>19.0</td>
<td>62.0</td>
<td>19.0</td>
</tr>
</tbody>
</table>

**Table 5:** The derived densities of each subtask when the Equal Slack strategy is adopted. The symbol \( \times \) represents that the relative deadline for a subtask is too small to complete its execution.

<table>
<thead>
<tr>
<th>Thread Number for ( s_{1,2} )</th>
<th>( \delta_{i,1} )</th>
<th>( \delta_{i,2} )</th>
<th>( \delta_{i,3} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \times )</td>
<td>( \times )</td>
<td>( \times )</td>
</tr>
<tr>
<td>2</td>
<td>0.075</td>
<td>0.832</td>
<td>0.075</td>
</tr>
<tr>
<td>3</td>
<td>0.053</td>
<td>0.710</td>
<td>0.053</td>
</tr>
<tr>
<td>4</td>
<td>0.046</td>
<td>0.635</td>
<td>0.046</td>
</tr>
</tbody>
</table>

On the other hand, if the Equal Slack (EQS) strategy [6, 11] is used to assign relative deadline for subtasks of a parallel task, the derived relative deadlines and densities are presented in Tables 4 and 5, respectively.

A feasible assignment exists when the parallelization level of the second segment \( s_{1,2} \) of \( \tau_i \) is equal to 2, i.e., executed by two threads. One subtask for \( s_{1,2} \) and \( \tau_3 \) are assigned to \( P_{\tau_1} \). The other subtask for \( s_{1,2} \) and \( \tau_4 \) are assigned to \( P_{\tau_2} \). The first subtask and the third subtask of \( \tau_1 \), \( \tau_2 \), \( \tau_5 \), \( \tau_6 \), and \( \tau_7 \) are assigned to \( P_{\tau_3} \).

For a general case, we can use the following equation to derive the total density \( \delta_{i,x} \) of subtasks for a task \( \tau_i \), i.e.,

\[
\delta_{i,x} = \sum_{j=1}^{3} \delta_{i,j,x} = \delta_{i,1,x} + x \delta_{i,2,x} + \delta_{i,3,x}
\]

\[
= \frac{c_{i,1}}{d_{i,1,x}} + x \frac{c_{i,2,x}}{d_{i,2,x}} + \frac{c_{i,3}}{d_{i,3,x}} + \frac{c_{i,1}}{C_1 + C_2} + \frac{x C_{i,2}}{C_{i,2} + \frac{T_i}{3} (T_i - c_{i,2} + c_{i,3})} + \frac{c_{i,3}}{C_{i,3} + \frac{T_i}{3} (T_i - c_{i,1} + c_{i,2} - c_{i,3})}
\]

We assume that \( C_1 = T_i - c_{i,1} - c_{i,3} \). Let define \( f(x) \) as the function of the WCET \( c_{i,2} \) of the segment \( s_{1,2} \) when the parallelism level is equal to \( x \). Then, the equation can be rewritten as follows:

\[
\delta_{i,x} = \frac{c_{i,1}}{c_{i,1} + \frac{T_i}{3} (T_i - c_{i,1} - c_{i,3})} + \frac{x C_{i,2}}{C_{i,2} + \frac{T_i}{3} (T_i - c_{i,1} - c_{i,2} - c_{i,3})} + \frac{c_{i,3}}{C_{i,3} + \frac{T_i}{3} (T_i - c_{i,1} - c_{i,2} - c_{i,3})}
\]

From the motivational example, we can observe that for a parallel task the EQS strategy can result in a lower total density of its subtasks compared with the EQF strategy. Hence, under the EQS strategy the tested task set has a higher probability to derive a feasible assignment. In the next section, we will present a systematic methodology to do assignment for a given task set.

### 3.2 Best-Fit Task Assignment based on the Equal Slack Strategy

Based on the example in the previous section, in order to derive a feasible (sub)task-to-processor assignment we should first consider to set proper relative deadlines for the subtasks of parallel tasks. Second, we will do the assignment for each task or subtask according to its density, which is defined as its WCET divided by its (derived) relative deadline. In this section, we will describe the proposed off-line Best-Fit based on Equal Slack (BFES) algorithm which consists of two steps: deadline setting and (sub)task assignment. At run time, the (sub)tasks assigned to each processor will be scheduled according to the EDF algorithm [12, 11].

Let \( T \) and \( M \) denote the task set and the processor set, respectively. \( T \) includes a sequential task subset \( T_s \) and a parallel task subset \( T_p \). The number of tasks in the \( T \) is \( N \) and the number of processors in \( M \) is \( M \). The procedure code of the BFES algorithm is depicted in Algorithms 1, 2,
Algorithm 1 Assign (sub)Tasks to Processors based on the Adopted Assignment Strategy

1: Function Assign_Task(\(T\))
2: Set \(T' = T\);
3: Set \(A_j \leftarrow \phi\) for each processor \(Pr_j\), \(1 \leq j \leq M\);
4: Set \(\Delta_j \leftarrow 0\) for each processor \(Pr_j\), \(1 \leq j \leq M\);
5: Set \(X_i \leftarrow 0\) for each task \(\tau_i\), \(1 \leq i \leq N\);
6: for each \(\tau_i\) in \(T\) do
7: if \(\tau_i\) is a sequential task then
8: \(X_i \leftarrow 1\);
9: else
10: Set_Deadline(\(\tau_i\));
11: \(x \leftarrow \text{Find_Minimum_Parallelism_Level}(\tau_i)\);
12: if \(x = -1\) then
13: return FALSE;
14: end if
15: \(X_i \leftarrow x\);
16: Let \(T_i\) be the subtask set for \(\tau_i\), which includes \(\tau_{i,1}\), \(\tau_{i,3}\), and \(x \tau_{i,2}\);
17: \(T' = T' \cup T_i\);
18: end if
19: end for
20: Sort all tasks in \(T'\) in a descending order of their densities;
21: for each \(\tau_i\) in \(T'\) do
22: \(p* \leftarrow \text{Find_Best_Processor}(\tau_{i,j}, \Delta)\);
23: if \(p* = -1\) then
24: return FALSE;
25: end if
26: Assign subtask \(\tau_{i,j}\) to processor \(p*\), i.e., \(A_{p*} \leftarrow A_{p*} \cup \tau_{i,j}\);
27: end for
28: return TRUE;
29: End Function

Algorithm 2 Set Relative Deadlines for Subtasks for Parallel Tasks Based on the EQS Strategy

1: Function Set_Deadline(\(\tau_i\));
2: for \(j = 1\) to \(M\) do
3: \(slk \leftarrow \frac{T_i - c_{i,1} - c_{i,2} - c_{i,3}}{3}\);
4: if \(slk > 0\) then
5: \(d_{i,1,j} \leftarrow c_{i,1} + slk\);
6: \(\delta_{i,1,j} \leftarrow \frac{d_{i,1,j}}{c_{i,1}}\);
7: \(d_{i,2,j} \leftarrow c_{i,2,j} + slk\)
8: \(\delta_{i,2,j} \leftarrow \frac{d_{i,2,j}}{c_{i,2}}\);
9: \(d_{i,3,j} \leftarrow c_{i,3} + slk\);
10: \(\delta_{i,3,j} \leftarrow \frac{d_{i,3,j}}{c_{i,3}}\);
11: else
12: \(d_{i,1,j} \leftarrow \infty\);
13: \(\delta_{i,2,j} \leftarrow \infty\);
14: \(\delta_{i,3,j} \leftarrow \infty\);
15: end if
16: end for
17: End Function

3. and 4. We will describe the purposes of the functions in the algorithm. We use the proposed off-line assignment algorithm to derive a feasible assignment for the task set before the system starts to execute the task set.

The major function of the proposed algorithm is the function Assign_Task(.) in Algorithm 1. First, we define \(T'\) as the new task set which is initially set as the sequential task subset. Then, the subtasks of each parallel task will be included into \(T'\). We define \(A = \{A_1, A_2, ..., A_M\}\) as an assignment, where \(A_j \in A\) represents the task subset \(A_j\) assigned to the processor \(Pr_j\). \(\Delta_j\) is defined as the sum value of total density of sequential tasks and the total density of the subtasks in the subset \(A_j\). We also denote \(X = \{X_1, X_2, ..., X_N\}\) as the parallelization level for a sequential task is always equal to 1.

The function Set_Deadline(.) in Algorithm 2 is invoked to set the relative deadlines for the subtasks of parallel tasks in \(T_j\). When the corresponding relative deadline is set for each subtask of a parallel task, the precedence constraints among the subtasks of each parallel task are translated into the timing constraints for the subtasks. Then, each subtask of a parallel task \(\tau_i\) has a relative deadline smaller than the period \(T_i\) of \(\tau_i\). The function uses the Equal Slack (EQS) strategy to set the relative deadlines for the subtasks when the second segment of \(\tau_i\) is under different parallelism levels. The number of parallelism levels is equal to the processor number \(M\). Besides, also we compute the densities of subtasks. Under the parallelism level \(j\) of the second segment of a task \(\tau_i\), if the slack \(\frac{T_i - c_{i,1} - c_{i,2} - c_{i,3}}{3}\) for each subtask of \(\tau_i\) is negative, the subtasks cannot complete their execution in time. Therefore, we set the densities of the subtasks as infinite.

For each parallel task \(\tau_i\), the function Find_Minimum_Parallelism_Level(.) is invoked to derive a proper parallelism level such that the density of each sub-task is no more than 1 and the total density of its subtasks is the smallest. Then, if the returned parallelism level \(x\) is not equal to \(-1\), the task \(\tau_i\) is transformed into \(x + 2\) subtasks, i.e., \(\tau_{i,1}, \tau_{i,3}, \) and \(x \tau_{i,2}\). The subtasks will be included into \(T'\). Note that for a sequential task its period is equal to its relative deadline, and its utilization and density are the same. Then, the (sub)tasks in \(T'\) are sorted in a descending order of their densities. Finally, the function incrementally deals with the (sub)tasks in \(T'\) according to the sorted order of the densities. A (sub)task is assigned to which processor based on the return value of the function Find_Best_Processor(\(\tau\)). The function finds the processor with the lowest enough available capacity for each (sub)task. If the function can not find a proper processor, it will return \(-1\). It represents that there exists no feasible assignment.

In the first loop of the function Assign_Task(.) in Algorithm 1, each parallel task \(\tau_i\) is tested and can translated into \((M + 2)\) subtasks in the worst case. The time complexity of the loop is \(O(NM)\) because there are \(N\) tasks and each task has \(M\) parallelism options to be tested. In the worst case, the number of subtasks in \(T'\) is equal to \(N \cdot (M + 2)\). Therefore, the time complexity of task sorting is \(O(MN(\log MN))\). Then, in the second loop there are \(N \cdot (M + 2)\) subtasks
to be assigned and one of $M$ processors is selected for each subtask. The time complexity of the loop is $O(NM^2)$. Finally, the run time complexity of the proposed BFES algorithm is $O(NM(M + \log NM))$, which comes from task sorting and the for loops in the functions, Assign_Task( ), Find_Minimum_Parallelism_Level( ), and Find_Best_Processor( ). $N$ and $M$ are the numbers of tasks and processor, respectively.

Algorithm 3 DERIVE A PROPER PARALLELISM LEVEL FOR A PARALLEL TASK

1: FUNCTION Find_Minimum_Parallelism_Level($\tau_i$)
2: Let $x$ be the minimum parallelism level and $x$ is initially set as $-1$;
3: Set EXIST $\leftarrow$ FALSE;
4: for $j = 1$ to $M$ do
5: if $(\delta_{i,j} \leq 1)$ and (EXIST = FALSE) then
6: $x \leftarrow j$;
7: EXIST $\leftarrow$ TRUE;
8: break;
9: end if
10: end for
11: if EXIST = TRUE then
12: for $j = 1$ to $M$ do
13: if $(\delta_{i,j} \leq 1)$ and $(\delta_{i,1,j} + x \times \delta_{i,2,j} + \delta_{i,3,j}) < (\delta_{i,1,1} + x \times \delta_{i,2,1} + \delta_{i,3,1})$ then
14: $x \leftarrow j$;
15: end if
16: end for
17: end if
18: return $x$;
19: END_FUNCTION

After a deadline assignment strategy is used to compute the relative deadlines for the subtasks of a parallel task $\tau_i$, each subtask executes with the new relative deadline. If all the subtask for $\tau_i$ meet their relative deadlines, then $\tau_i$ will meet its original deadline. In the following, we propose the properties for the proposed algorithm. Based on the properties, each subtask of $\tau_i$ can be assigned to any processor if the sum value of the total density of (sub)tasks assigned to the processor and the density of the subtask is no more than 1. We assume that the second segment $s_{i,2}$ of a parallel task $\tau_i$ is executed by $x$ threads, the relative deadlines of the subtasks, $\tau_{i,3}$, $x$ subtasks for $s_{i,2}$, and $\tau_{i,3}$, of a parallel task are set with a deadline assignment strategy, e.g., EQF or EQS.

**Theorem 1.** If the total density of the subtasks $\tau_{i,1}$, a subtask for $s_{i,2}$, and $\tau_{i,3}$, is no more than 1, then they could be assigned to a processor with available capacity. The three subtasks are schedulable according to the EDF algorithm.

**Proof.** With the adopted deadline assignment strategy, the precedence constraints are translated into the timing constraints. When the total density of the three subtasks $\tau_{i,1}$, one subtask for $s_{i,2}$, and $\tau_{i,3}$ is no more than 1, the subtasks can execute on a processor. Their deadlines will be met when the subtasks are scheduled with the EDF algorithm.

**Theorem 2.** If the total density of $y \leq x$ subtasks for $s_{i,2}$ is no more than 1, then they could be assigned to a processor with available capacity. The $y$ subtasks are schedulable according to the EDF algorithm.

4. PERFORMANCE EVALUATION

4.1 Experiment Setup and Performance Metrics

The purpose of this section is to provide the performance evaluation of our proposed algorithm, referred to as the Best-Fit based on Equal Flexity (BFES) algorithm. A simulation model was investigated for the performance of the proposed algorithm. We compared the performance of the BFES methodology with the First-Fit based on Equal Slack (FFES), Worst-Fit based Equal Slack (WFES), Worst-Fit based on Equal Flexity (WFFE), and Best-Fit based on Equal Flexity (BFIF) strategies [2].

The tested task set is generated with three factors: parallelization overhead $\alpha$ for parallel tasks, the number of parallel tasks $N_p$ in the task set, and workload ratio $\gamma$ of the system. The performance of the BFES scheme was simulated when the processor number was set as 8. The workload of periodic tasks in the investigated system was denoted as $U$. We varied the ratio $\gamma$ between 0.2 and 0.9 with increments of 0.1. For example, when the $\gamma$ value was set as 0.6, the workload $U$ was equal to 4.8 ($= 0.6 \times 8$). The workload consisted of sequential tasks and parallel tasks. The number $N_p$ of parallel tasks in a tested task set was set from 1 to 4. The utilization of each sequential task was uniformly selected in the range of $[0.01, 0.20]$ while that of each parallel task was uniformly selected in the range of $[0.1, 0.30]$. The period of each task was randomly generated within the range of $[100, 3000]$. Each parallel task $\tau_i$ has three segments, the first segment $s_{i,1}$, the second segment $s_{i,2}$, and the third...
segment \( s_{i,3} \) is a parallel segment, and \( s_{i,1} \) and \( s_{i,3} \) are sequential segments. Due to the limitation of the processor number, \( s_{1,2} \) of \( \tau_i \) has eight parallelism options. The maximum parallelization level equal to eight while the minimum one is equal to one. Let \( c_{i,2,1} \) be the worst-case execution time (WCET) of \( s_{i,2} \) executed under the parallelism level 1. \( c_{i,1} \) and \( c_{i,3} \) are the WCETs for \( s_{i,1} \) and \( s_{i,3} \), respectively. We assumed that \( c_{i,1} = c_{i,3} \). The ratio of \( c_{i,2,1} \) to \( (c_{i,1} + c_{i,3}) \) was uniformly selected in the range of [95%, 99%]. The worst-case execution time (WCET) \( c_{i,2,j} \) under parallelization level \( j \), where \( 2 \leq j \leq M \), for the parallel segment was generated based on [10], i.e.,

\[
c_{i,2,j} = \frac{c_{i,2,1}}{j} + \alpha(c_{i,2,1} - \frac{c_{i,2,1}}{j}).
\]

The parallelization overhead \( \alpha \) value was set in the range of [0, 1.0] with increments of 0.1.

1,000 task sets were generated for each generation factor setting. Therefore, each experimental simulation result was an average value over these 1,000 independent task sets. The performance metric measuring the task sets was the normalized schedulability ratio. We used the WFES strategy as a base line to compare against the other compared algorithms. \( N_{WFES} \) is defined as the number of the tested task sets for which the WFES derived a feasible task assignment. \( N_{Alg} \) is denoted as the number of the tested task sets for which the \( Alg \) strategy derived a feasible task assignment. The normalized schedulability ratio is defined as the ratio of \( N_{Alg} \) to \( N_{WFES} \).

### 4.2 Experimental Results

Figure 2 depicts the experiment results when the processor number \( M \) is equal to 8 and the workload ratio \( \gamma \) is equal to 0.6. Figures 2(a), 2(b), and 2(c) indicate the normalized schedulability ratio of tested task sets when the number \( N_p \) of parallel tasks in the task sets is equal to 1, 2, and 3, respectively. Note that when the schedulability ratios of both of the WFES algorithm and the compared algorithm \( Alg \) are equal to 0, the normalized schedulability ratio for the algorithm \( Alg \) is equal to 0. When \( M = 8 \) and \( \gamma = 0.6 \), the total utilization of tasks in the system is 4.8. A parallel task with the utilization more than 1.0 cannot finish its execution within the deadline and must be parallelized. Let \( \delta_{EQS,x}^\tau \) and \( \delta_{EQF,x}^\tau \) be the summing value of the densities of all the subtasks for \( \tau_i \) under the EQS and EQF strategies, respectively, when the parallelization level is equal to \( x \). Under the EQS strategy, a smaller parallelization level \( x \) value implies a larger \( \delta_{EQS,x}^\tau \) value. Under the EQF strategy, \( \delta_{EQF,x}^\tau \) is not directly proportional to the parallelization level. Besides, when the parallelization level \( x \) is small, \( \delta_{EQF,x}^\tau \leq \delta_{EQS,x}^\tau \). However, when \( x \) becomes large, \( \delta_{EQF,x}^\tau \geq \delta_{EQS,x}^\tau \). Hence, the density of each subtask of \( \tau_i \) no more than 1 and \( \tau_i \) has the smallest summing value of the densities of all the subtasks when \( x = 2 \).

From Figure 2(a) we can observe that the compared algorithms have the same performance results when the parallelization overhead \( \alpha \) value is less than or equal to 0.2. The reason is that when the \( \alpha \) value is small, the incremental parallelization overhead is small such that there still exists a feasible task assignment for each task set. We can also find that when \( \alpha \geq 0.3 \), the assign algorithms, i.e., WFES, FFES, and BFES, based on the EQS strategy outperform that based on the EQF strategy. It is because that a larger \( \alpha \) value implies a larger \( \delta_{EQF,x}^\tau \) value. The incremental density of a parallel task makes that the algorithms can not derive a feasible assignment. The normalized schedulability ratio of the proposed BFES algorithm is slightly larger than that of the WFES algorithm and is the same as that of the FFES algorithm. The reason is that the BFES and FFES algorithms assign a (sub)task to the processor with the smallest available capacity and leave a larger available capacity on another processor. Figure 2(b) indicates the performance when the number \( N_p \) of parallel tasks is equal to 2.

From Figure 2(b), we can observe that the performance trend is similar to that in Figure 2(a). However, the algorithms based on the EQF strategy have no feasible task assignments when \( \alpha \geq 0.2 \). This is because a larger \( N_p \) value implies that a larger increment contribution to \( \delta_{EQF,x}^\tau \). Therefore, it is hard for the algorithms to derive a feasible assignment for a tested task set. The algorithms based on the EQS strategy can not also find feasible assignments when \( \alpha \geq 0.7 \). Figure 2(c) indicates the performance when the number \( N_p \) of parallel tasks is equal to 3. The performance results are similar to that in Figures 2(a) and 2(b). The compared algorithms can not derive any feasible assignment when \( \alpha \geq 0.3 \). When the system workload is the same, the compared algorithms become hard to derive a feasible assignment if there are more parallel tasks in the task set.
Note that when $N_p = 4$, the normalized schedulability ratios of the compared algorithms are equal to 0 and the performance results are not shown in this paper.

Figure 3 illustrates the experiment results when the processor number $M$ is equal to 8 and the number $N_p$ of parallel tasks is equal to 1. Figures 3(a) and 3(b) indicate the normalized schedulability ratio of tested task sets when the parallelization overhead $\alpha$ value are equal to 0.7 and 0.8, respectively. We can observe that the algorithms based on the EQS strategy outperform the algorithms based on the EQF strategy. The reason is similar to that of Figure 2(a). A larger parallelization overhead $\alpha$ value implies that a parallel task has a larger total worst-case execution times for its second segment when the task is parallelized. Under the EQS strategy the summing value of the densities of all the subtasks of a parallel task $\tau_i$ at a low parallelism level, i.e., 2 or 3, is less than that under the EQF strategy. Figure 4 shows the normalized schedulability of the algorithms when the workload ratio $\gamma$ is equal to 0.7, and the parallelization overhead $\alpha$ is equal to 0.1. A larger $N_p$ value implies that a larger increment contribution to $\delta_{EQF,x}^{2}$. Therefore, it is hard for the algorithms to derive a feasible assignment for a tested task set when $N_p = 4$, each compared algorithm derives no feasible assignment for each tested task set.

From the above experimental results we can conclude that for a tested task set the algorithms based on the EQS strategy have higher probabilities to derive a feasible task assignment compared with the algorithms based on the EQF strategy. Hence, we can conclude that the proposed BFES algorithm can better performance than the other compared algorithms.

5. CONCLUSIONS

This paper addresses a scheduling problem where the system with multiprocessor has sequential tasks and parallel tasks. Because a parallel task can not meet its deadline without parallelization, we propose the Best-Fit based on Equal Slack (BFES) algorithm to solve such a scheduling problem. Based on the Equal Slack strategy the BFES algorithm derives a proper parallelization level for each parallel task in a tested task set and computes the densities of the subtasks of parallel tasks. Then, each (sub)task is assigned to a processor. A series of simulation experiments were conducted to show the capability of the proposed algorithm, for which we have very encouraging results.

For the future work, we can investigate the scheme to consider if each parallel task has more than one parallel segment. Besides, we try to extend the algorithm to the system with multiple heterogeneous processors.

6. ACKNOWLEDGMENTS

This study was supported by Ministry of Science and Technology of R.O.C., project No. MOST 104-2628-E-390-002-MY3 and 104-2628-E-025-001-MY2.

7. REFERENCES


### ABOUT THE AUTHORS:

<table>
<thead>
<tr>
<th>Chin-Fu Kuo</th>
<th>Yung-Feng Lu</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Chin-Fu Kuo" /></td>
<td><img src="image2.png" alt="Yung-Feng Lu" /></td>
</tr>
</tbody>
</table>

**Chin-Fu Kuo** received his BS and MS degrees in the Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan, R.O.C. in 1998 and 2000, respectively. He received the Ph.D. degree in the computer science and information engineering, National Taiwan University, in Taipei, Taiwan, R.O.C. in 2005. He joined the Department of Computer Science and Information Engineering (CSIE), National University of Kaohsiung (NUK), Kaohsiung, Taiwan, R.O.C. as an Assistant Professor in 2006. Currently, he is an Associate Professor in CSIE of NUK, Kaohsiung, Taiwan. His research interests include real-time process scheduling, resource management, cloud computing, and network QoS.

**Yung-Feng Lu** received the B.S. and M.S. degrees from the Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C., in 1998 and 2000, respectively, and the Ph.D. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, R.O.C. in 2010. He received Best Paper Award from the IEEE/IFIP International Conference on Embedded and Ubiquitous Computing (EUC) in 2008. He joined the Department of Computer Science (CSIE) and Information Engineering at National Taichung University of Science and Technology (NTCUST), Taichung, Taiwan, R.O.C. as an Assistant Professor in 2010. Currently, he is an Associate Professor in CSIE of NTCUST. His research interests include networking, embedded systems, databases, storage, neural networks, cloud computing, internet of things, and information security.
ABSTRACT

Built-in data structures are a key contributor to the performance of dynamic languages. Record data structures, or records, are one of the common advanced, but not easily optimizable built-in data structures supported by these languages. Records may be used in an object-oriented fashion or to implement object orientation itself.

In this paper, we analyze how records are used in different applications in the Scheme dialect Racket. Based on the data obtained, we suggest the application of existing optimization techniques for records and devise a new one for immutable boolean fields. Most of them can be applied to a wide range of record implementations in dynamic languages. We apply these optimizations to records in Pycket, an implementation of Racket. With one exception, micro-benchmarks show a two- to ten-fold speed-up of our implementation over plain Racket.

CCS Concepts

- Information systems → Record and block layout;
- Software and its engineering → Data types and structures;
- Classes and objects;
- Just-in-time compilers;

Keywords

Record data structures; Objects; Racket; Optimization

1. INTRODUCTION

For programming language implementations, performance is often key and, among other aspects, built-in data structures contribute to the overall performance of a language implementation. The lack of optimization of built-in data structures may result in poor performance and increased memory consumption of dynamic languages [2, 18]. In the context of modern virtual machine (VM) development frameworks, such as RPython, some data structures, such as collections [5], are already in the focus of research.

2. BACKGROUND

Record data structures, or records, are collections of named fields of heterogeneous values. Records may form a type, instances of record
The example in listing 1 contains two structure instances: a `values`. Certain properties can be used to make structure instances
arrays. Individual arrays
(employee? worker)
11
(employee-salary worker)
14
(workstation? john-station)
21
#s(workstation "station01" "fd23:5e15:aa18::2" 5))
20

be used to update the field. The accessor
employee-salary
salary
person
fined in line 3, makes use of structure hierarchies — it is a sub-type
(named "John Smith") in the "Developer" position (employee? worker)
11
(employee? customer)
12
(employee? worker)
12
(workstation? john-station)
20

Listing 1: Racket structures using structure hierarchies, explicit
mutability, callable structures, and prefabs.
1 (struct person (name))
2 (define customer (person "Sam Adams"))
3 (struct employee person (position [salary #:mutable]))
4 #:property prop:procedure
5 (lambda (self) (* (employee-salary self) 0.146)))
6
7 (define worker (employee "John Smith" "Developer" 50000))
8 (person? 0) ; -> #f
9 (person? customer) ; -> #t
10 (person? worker) ; -> #t
11 (employee? customer) ; -> #f
12 (employee? worker) ; -> #t
13
14 (set-employee-salary! worker 55000)
15 (employee-salary worker) ; -> 55000
16
17 (worker) ; -> 7300	
18
19 (define john-station
20 ;;$(workstation "station01" "fd23:5e15:aa18::2" 5))
21 (struct workstation (name ip age) #:prefab)
22 (workstation? john-station) ; -> #t

types are typically of equal size—all in contrast to data structures like
arrays that are collections of typically indexed fields of homogenous
values. Array-like data structures do not form types. Individual arrays
may differ in size. Moreover records may have various additional
features, which may differ between programming languages.

2.1 Structures in Racket
Racket [12] is a dynamically typed, multi-paradigm programming
language from the Scheme-family [21]. Racket differs from Scheme
in certain aspects such as immutable-by-default lists, built-in support
for design by contract [17], or a more complex record data structure
concept called structures (or structs), providing features beyond the
mere ability to store values in their fields.

Racket structure types can form hierarchies, supporting inheritance.
Structures in Racket are immutable by default, but can be explicitly
declared to be partly or fully mutable. Structure type properties allow
to store arbitrary data inside the structure type. However, typically
properties are used for procedures that work on a structure’s field
values. Certain properties can be used to make structure instances
callable; these structures can then act like procedures. Other prop-
erties denote structures as transparent, allowing run-time reflection
on a structure’s internals. Racket also supports a shortcut form of
structures for literal specification of structures before the formal in-
troduction of their structure type. These structure types are called
previously fabricated structure types (prefabs).

The example in listing 1 contains two structure instances: a person
named “Sam Adams”, bound to customer in line 2. The correspond-
ing structure type person is defined in line 1 as structure with one
field, name. The predicate person? further down confirms this. The
second structure instance bound to worker in line 7 is an employee
named “John Smith” (employee? worker) who earns 50 000 money (salary). The structure type employee, de-
defined in line 3, makes use of structure hierarchies—it is a sub-type
of person and inherits its name field. Moreover, it has a mutable field
salary. Hence, the mutator set-employee-salary! further down can
be used to update the field. The accessor employee-salary can be
used to retrieve the stored value. Then, the structure type has a prop-
erty named prop:procedure that is bound to a procedure. That way,
calling the worker structure instance in the last line results in the
procedure to be called with this instance and computes the amount of
medical insurance fee based on the salary and the fixed rate. Lastly,
john-station is defined as a prefab structure in line 20, without the
need (but possibility) to define the structure type workstation before-
hand. However, such a structure type can be defined after the fact,
as in the line following. The predicate workstation? confirms that,
indeed, john-station is of the expected structure type.

2.2 Structures and Objects
Scheme is a multi-paradigm language family that is probably best
known for its functional aspects. However, object-orientation is not
only possible to implement and use, for example with Common Lisp Object System (CLOS) implementations such as TinyCLOS, in
Racket an object-oriented (OO) implementation is readily available
with the racket/class standard library. It provides class-based object
orientation with message passing, mixins, and traits [13]. This system
is implemented in terms of Racket structures; every class is also a
structure type, every object is a structure instance. While it would
have been possible to focus solely on the object-oriented part of
Racket, considering all structures instead benefits the implementation
of object orientation as well as other parts of Racket.

Racket structures actually can directly be used in an object-oriented
fashion—at the loss of message passing and run-time polymorphism
compared with the library implementation of object orientation.
However, other object-oriented fundamentals, such as instance identity,
encapsulation, abstraction, and even object behavior are already
present in Racket’s base structures and also justify an investigation
under an object-oriented point of view.

3. STRUCTURE USAGE IN RACKET
Racket structures are a powerful data structure with broad applica-
bility. They are widely used in Racket packages3 and projects on
GitHub7. Structures are essential for the Racket contracts implemen-
tation. In this section, we investigate how structures are actually used
in different Racket applications. We perform a static and dynamic
analysis of existing applications to identify the typical size of struc-
tures, types used within structures and the frequency of mutation.

We choose five Racket applications from different domains including
development tools, text analysis, mathematics, and games. I Write
Like3—one of the biggest Racket applications—is a web application
that analyses the style of a given text by comparing with styles of many
famous writers. This application represents a heavy text analysis
application. The markdown parser application6 is a simple parser for
markdown formatted text that is used in many other Racket projects as
a library. Racket CAS5 is a simple computer algebra system for Racket
with a good built-in test set. 20486 is a Racket implementation of a
famous puzzle-game with numbers. Finally, DrRacket is a feature-
rich Racket integrated development environment (IDE), which is
widely used by Racket-programmers.

3 http://pkgs.racket-lang.org (visited 2015-12-05)
4 https://github.com/search?q=language%3Aracket (v. 2015-12-05)
5 https://github.com/coding-robots/iwl (visited 2015-12-05)
6 https://github.com/soegaard/racket-cas (visited 2015-12-05)
7 https://github.com/greghendershot/markdown (visited 2015-12-05)
8 https://github.com/danprager/racket-2048 (visited 2015-12-05)
3.1 Static Analysis

We perform a static source code analysis of the Racket v6.2.0.4 standard library comprising 4,812 Racket source code files. We track the number of immutable and mutable fields and super types per structure.

3.1.1 Results

Of all the source files, 11.6% contain all 1,765 structure type definitions (cf. Table 1), 31.9% with super-types. Structures have $2.3 \pm 2.6$ fields on average, with a median of 2. The largest structure from the Racket library has 37 fields. 91.6% of all structure types are immutable. Structures with mutable fields tend to be larger (maximum: 37, mean: $4.55 \pm 4.56$) than all-immutable structures (maximum: 18, mean: $2.10 \pm 2.17$). The distribution is shown in Figure 1.

The statically determined number of structure types in the applications analyzed is comparatively small; together, they define 22 structure types with at most 5 fields (average $1.64 \pm 1.26$, median 1), all immutable. We refrain from plotting the distribution.

3.2 Dynamic Analysis

We instrumented the structure implementation in Racket to track the creation process of structure types, structure instances, the amount and types of structure field values, and the frequency of mutate operations. Our analysis reports the total usage of structures including the Racket core.

3.2.1 Results

Refining the static analysis, about 85% of all fields used are immutable, with DrRacket being an outlier with about 61% of immutable fields. Structure instances have $1.62 \pm 4.56$ fields on average with a median of 1. The number of instances of each structure type depends heavily on the specific application, ranging from 200 to 1500 in our tests. The number of mutations varies even more.

Although structures in Racket are typically used monomorphic, that is the data type of values stored in a field does not change, some instances’ fields are used with values of more than one data type (non-monomorphic). The amount of structures containing at least one non-monomorphic field is between 5% and 15%.

The distribution of field types is homogeneous as illustrated in Figure 2. The most common data type used in structure field type is boolean. Up to 70% of booleans have the value #f (false), which is used in up to 88% as a placeholder default value for other data types, such as procedure. Procedures are also used widely, to the extent that some structures only contain exactly one procedure — such procedure-containers are often used as super-types for other structures. Strings, mutable and immutable, pose the most user-faced data type in field types while symbols and the syntax type (used by the Racket macro system) are more system-faced, or even meta-level types used in structures. Non-scalar field types, such as pairs and lists, and other structures are common as field types, too. Other types have a collective share of about 10%.

Despite our initial assumption, integers are not very common, except for the 2048 game that heavily uses integers and floats. Other applications use numbers significantly less frequently. To show this, we separated 2048 in Figure 2.

We found only few common data type collocation patterns in structures, despite the homogeneous field type distribution. Such patterns...
We first present a most simple approach to realizing Racket structures, which is relatively small and contains super reference fields in its storage. Such an implementation is typical for objects in languages, for example Squeak / Smalltalk [15]. This approach loses the redundant super-instance / sub-instance tandem and hence is not considered best practice but rather serves as a baseline for the optimizations to come.

This approach anticipates the Racket way to access inherited fields. For example, to access the name of worker in our example, the native accessor behavior will be called with an offset 0 and the structure type person, but to access position, it will be called with an offset 0, too, but this time with the structure type worker. However, certain performance improvements already become apparent: super-instances never exist solitarily but always together with their sub-instances. Furthermore, they duplicate the hierarchy information already available in the type.

### 4.2 General Optimizations

We first consider and apply existing optimizations for records and similar structures. The combined approach is illustrated in Figure 4.

#### 4.2.1 Flat Structure

A flat structure collapses the semantical hierarchy of record objects and represents every record with only one object that combines all fields in its storage. Such an implementation is typical for objects in OO languages, for example Squeak / Smalltalk [15]. This approach loses the redundant super-instance/sub-instance tandem and hence improves memory consumption.

A flat structure saves memory by removing redundant record objects. A simple, straight-forward record object has a size of four machine-words multiplied by the number of super-types, one for the header.

---

**Figure 2:** Most frequent field types (left) and most frequent combinations of field types (right) in Racket applications

**Figure 3:** Direct mapping approach representation of worker

**Figure 4:** Forward direct-mapping approach representation of structures in Racket applications.
Record data structures in dynamic programming languages can contain fields of arbitrary types. A usual way to implement storing of different data types together is boxing, that is, an allocation of all field values on the heap with a common header. Boxing simplifies the implementation of dynamic programming language significantly, because all different objects obtain a common simple representation. Nevertheless, boxing is not always efficient.

For example, to store an integer in C, only one machine-word is typically needed. For comparison, it typically needs at least three words to store a boxed integer in a dynamical language running in the VM (one to store the type of object and the last one to store the particular value of integer, and typically one for the GC) [5]. This problem gains importance when many objects are stored together in records.

One solution is to store the record field values unboxed, saving the type information separately once for many objects. The object where this type information is stored is called field type. However, a mutation of a field to a new value with another type involves dereferencing of the object where the original type information was stored. To avoid this, the type information needs to be stored in the record’s object representation. However, every object needs to store the corresponding type information, which may not be used for all fields in the object. Figure 4: Flat structure, inlined fields, and mutable salary field, wrapped into a typed cell with an unboxed value.

and three for references to the type, super instance, and storage for each record object in the hierarchy. Additionally, every record object typically has a storage that takes two additional words (typically one for the length and one with special purpose, commonly for the garbage collector (GC)). Records with a flat structure do not have super-instances, which saves one word for the reference to the super-type and six words for every super-type of every record instance, that is four word for the record and two for the storage. Assuming that about 31.9% of Racket’s structures have one super-type (cf. section 3), for n structure objects, that saves

$$n \cdot (6 \cdot 0.319 + 1)$$  \hspace{1cm} (1)

words in Racket on average.

Nevertheless, records with a flat structure make the implementation of the native accessing behavior more complex. The per-structure-type indices now have to be mapped to the absolute index into the record’s storage. These indices do not change over time and, therefore, a static mapping for each field can be calculated in advance.

4.2.2 Inlining

The direct mapping approach contains an indication between a structure’s representation entity and the actual storage for the structure’s fields. This eases the implementation of the representation entity, for example as instances of a structure class. This additional hop, however, can be cause for performance bottlenecks, as every field read has to traverse the indirection. A best practice is to fuse records and their storage, improving execution time performance by reducing costs of object allocation and pointer dereference. Implementations like the Squeak VM or the Java Virtual Machine (JVM) do this for their object representation.

Arbitrarily large structures may, however, slow down the overall allocation performance and hamper GCs. While Racket structures may have up to 32,768 fields\(^7\), the actual amount of structure fields used in Racket is typically low; between one and two fields on average (cf. section 3). Hence, we propose to limit inlining to only few fields and store larger records with a separate storage, as done in the PyPy implementation of Python [20].

Field inlining reduces the complexity of data access operations by removing a one additional access hop. This optimization releases three words for every structure instance, that is one word for a storage reference and two words for a storage array (length and special/GC word, as above). For n structures, that saves

$$n \cdot 3 - \text{sizeofSpecializedClasses}$$  \hspace{1cm} (2)

words, where the sizeofSpecializedClasses indicates the memory needed for specialized record classes containing a certain amount of fields. If the total amount of fields is greater than a predefined limit, fields are stored in the storage without changes. This avoids the creation of a big amount of record classes, which may not be used at all. The inlining technique has two important advantages. First, records with inlined fields take less space than records with a storage. But even more, inlining is crucial for optimizing the access to the record fields, because it avoids an indirection to a separate storage.

4.2.3 Unboxing and mutability separation with cells

Field typing reduces the complexity of data access operations by removing a one additional access hop. This optimization releases three words for every structure instance, that is one word for a storage reference and two words for a storage array (length and special/GC word, as above). For n structures, that saves

$$n \cdot 3 - \text{sizeofSpecializedClasses}$$  \hspace{1cm} (2)

words, where the sizeofSpecializedClasses indicates the memory needed for specialized record classes containing a certain amount of fields. If the total amount of fields is greater than a predefined limit, fields are stored in the storage without changes. This avoids the creation of a big amount of record classes, which may not be used at all. The inlining technique has two important advantages. First, records with inlined fields take less space than records with a storage. But even more, inlining is crucial for optimizing the access to the record fields, because it avoids an indirection to a separate storage.

Field inlining reduces the complexity of data access operations by removing a one additional access hop. This optimization releases three words for every structure instance, that is one word for a storage reference and two words for a storage array (length and special/GC word, as above). For n structures, that saves

$$n \cdot 3 - \text{sizeofSpecializedClasses}$$  \hspace{1cm} (2)

words, where the sizeofSpecializedClasses indicates the memory needed for specialized record classes containing a certain amount of fields. If the total amount of fields is greater than a predefined limit, fields are stored in the storage without changes. This avoids the creation of a big amount of record classes, which may not be used at all. The inlining technique has two important advantages. First, records with inlined fields take less space than records with a storage. But even more, inlining is crucial for optimizing the access to the record fields, because it avoids an indirection to a separate storage.
a field types object and even a creation of a new field types object if no proper field types object exists.

Assuming that structures have 1.6 fields on average and the part of structures with homogenised fields is about 85 %, for n structures, this optimization saves

\[ n \cdot 2 \cdot 1.6 \cdot 0.85 \cdot 0.44 - n - sizeOfFieldTypes \approx 0.2n \]

words in Racket, where sizeOfFieldTypes is the size of field types objects containing the type information of fields. That is, this optimization is not very efficient for small records in Racket. Furthermore, unboxing brings some memory overhead because of storing field types pointers and increases the implementation complexity. This overhead may lead to the negative optimization effect in the worst case, when records contain boxed values only.

The complexity of field types to support mutability can be alleviated by partially boxing the mutable content of structures. Also, implementations can take advantage of the fact that Racket structures have mostly immutable fields. Moreover, if all fields of a structure were always immutable, better optimizations would be imaginable; especially just-in-time (JIT) compilers that use tracing or partial evaluation could benefit. Combining this, we propose to treat all structures as immutable and use an indirect object, called cell, for the few fields that are actually mutable. Changing the value of a field no longer affects the structure itself but rather delegates the change to the cell representing the mutable field, as can be seen in Figure 6. That way, the maintenance of field types is completely absorbed into managing cells, which may or may not be typed. This technique is common in Lisp and Scheme applications, among others. As the mutability of fields is a property of a structure’s type, wrapping objects in cells can efficiently be done at structure allocation.

Using cells implies an inherent memory and access time overhead. However, as most fields are used monomorphic, we can specialize cells to typed cells, which store a type and an unboxed value. They can change their type field dynamically upon mutation. Thus, if a mutable record field belongs to a known type, such as integer or float, a typed cell stores its value unboxed, reducing the cell’s overhead.

We obtain an optimization that (a) improves execution time for access to mutable fields and (b) can reduce memory consumption for larger structures. However, the general structure size is small in Racket, so we propose to just use the cell part without the general unboxing.

4.3 Immutable Boolean Field Elision

Booleans are the most frequent field type in Racket structures. However, up to 70 % of boolean fields have the value #t. Knowing that most (up to 85 %) fields are actually immutable, a high number of fields in Racket structures hence consist of immutable boolean fields (IBFs).

It seems feasible to actually not store this information as a field value per se. Instead of storing both positions and values of the boolean fields, we use an indicator to denote all positions of IBFs within a structure, effectively eliding the immutable #t values; we call this immutable boolean field elision (IBFE). This indicator might be implementation specific; but in the same way structures that contain mutable fields or unboxed fields must be communicated to the runtime, IBFs can be communicated similarly, be it tagging, header bits, or class-based indication as in Figure 7, to name a few. It is crucial that all possible combinations of IBFs for an arbitrary record instance are present as indicators at structure allocation time. For example a record class with three fields, all immutable, that gets instantiated with an #t value on position two could use an implementation class that treats position two specially by not providing storage for it (cf. Figure 7). That implementation class would act as IBF indicator. Note that the #t value is not treated specially by immutable boolean field elision (IBFE), as are #f values in mutable fields. These are stored as if IBFE was not present at all.

The boolean optimizations saves memory by reusing immutable false values. Assuming that structures have an average size of 2.3 fields, 26 % of all fields are booleans and 70 % of booleans are false, and also that 85 % of fields are immutable in Racket, for n structure objects, this saves

\[ n \cdot 2.3 \cdot 0.26 \cdot 0.7 \cdot 0.85 - sizeOfSpecializedClasses \approx 0.36n \]

words in Racket on average, where sizeOfSpecializedClasses indicates the required memory for pre-defined structure classes with false fields. Although this optimization may have less positive impact on memory consumption on average, it does not add memory overhead for records in the worst case as unboxing with field types would. For extreme case, where every record has one immutable field with a value false, the saving would be approximately n.

Using IBFE, memory for immutable #f values can be saved at the expense of providing a large enough number of IBF indicators, which poses a trade-off. Applications with only few IBFs and large structures
We implemented the presented optimizations in Pycket, a Racket implementation using the RPython toolchain and its meta-tracing JIT compiler.

5. STRUCTURES IN PYCKET

We implemented the presented optimizations in Pycket, a Racket implementation using the RPython toolchain and its meta-tracing JIT compiler.

5.1 RPython and Pycket

RPython [4] is a framework for implementing interpreters, consisting of a type-inferenceable (“restricted”) subset of Python and a toolchain that translates an interpreter written in the RPython language into an efficient VM. Lower-level VM features, such as GC, object layout, and a meta-tracing JIT compiler are inserted automatically during the translation process. RPython was used for efficient implementations of several dynamic languages including Python [1], Prolog [7], and Smalltalk [6].

5.1.1 Meta-tracing

Typically, tracing JIT compilers optimize the executable program directly. For the meta-tracing JIT, the executable program is itself an interpreter running a user’s program code. In other words, the meta-tracing JIT operates on a representation of the interpreter. In order to produce efficient VM with RPython, the interpreter needs some hints from the developer, to help the tracing JIT to identify loops in the interpreted program, and perform other optimizations.

The RPython JIT compiler records operations executed by the interpreter running a user’s program. The produced linear sequences of machine code are called traces and they are only recorded for loops, which were performed more than a certain number of times. Only operations executed by the interpreter for one iteration of the loop are recorded by the tracing JIT compiler. The tracing process then optimizes the machine code instruction sequence and generates new machine code that is used for next iterations of the loop.

Because of the linearity of sequences, the trace represents only one of the potential execution flows. To provide the correctness of program execution, the JIT inserts guards, special instructions, which detect when the program execution conflicts with the trace and return control back to the interpreter.

5.1.2 Pycket

Pycket8 is an implementation of Racket using the RPython toolchain and based on the control, environment, and continuation (CEK) abstract machine [11]. Using the CEK machine eases the implementation of some more complex features of Racket, such as proper tail calls, first-class continuations, and multiple return values [3]. It is already competitive with the best existing ahead-of-time (AOT) Scheme compilers, particularly on safe, high-level, generic code [8]. However, it is not yet feature-complete and in particular had no structure support prior to this work.

5.2 Optimization Steps

Practically all implementations of record-like data structures skip the step Direct Mapping Approach described in section 4.1. However, for evaluation purposes, we included a direct-mapping-based implementation all following optimizations are applied to. Accordingly, all structure types are implemented as instances of an RPython-level class (W_StructType) and all structures as instances of a distinct class (W_Struct or its subclasses) with a reference to the structure type, a references to a storage for the fields, and possibly a reference to its super-instance.

5.2.1 Flat Structure

For a flat structure, a structure instance no longer refers to its super-instances but assumes all their former fields. However, the positions of all fields in the structure type hierarchy have to be mapped to the absolute fields positions to retain data access semantics: the language-level accessor and mutator procedures handle field indices relative to their respective type and respective to the top-most super-type. The absolute offsets do not change, however, and hence are calculated once during the structure type initialization and marked as immutable. This allows the JIT compiler to remove most field-position related calculations at runtime.

5.2.2 Inlining

The inlining optimization changes the data layout of structures by fusing structures and their storages together. This optimization requires the creation of modified structures, which may contain field values as attributes.

To inline fields into the structure instance, several specialized structure classes exist that each represent structures of a certain size. Following PyPy’s example, only up to 10 fields are actually inlined; larger structure instances still use a separate storage. Therefore, 12 implementation classes for structures are provided. The decisions which particular class is used for a structure instance is made at runtime as part of the instantiation process. Thus, if a new structure does not exceed the limit, one of the specialized implementation classes is chosen, and field values are saved in the structure’s attributes directly.

5.2.3 Typed Cells for Mutability Separation

Cells allow to keep all structure fields immutable by wrapping all mutable fields into cell objects. Cells stay immutable itself as a part of the structure, but may change their content.

The concept of a typed cell was already available in Pycket before introducing structure support and has been used for mutable globals and environment optimization, to name a few. Pycket cells store their values unboxed using storage strategies [5]. If a matching strategy exists, a cell stores its value unboxed, for example integer and float values. Otherwise, cells use a general strategy and store values boxed.

Hence, for structure support, upon creation of a structure instance, all mutable fields — which are known in advance — are wrapped by cells and all of the structure instance’s actual fields stay immutable. Also, all accessor and mutator behavior has been adapted to use the cells to unwrap and wrap valued automatically.

5.3 Eliding Immutable Boolean Fields

To benefit from immutable boolean fields, we suggested immutable boolean field elision (IBFE) in section 4.3. We chose to use the structure implementation class to represent the IBF indicators. As RPython does not support creation of RPython-level classes at runtime, all necessary indicators have to be generated in advance, before translation. However, a very high number of IBF classes can severely slow down allocation and possibly start-up time. Therefore, we assume an upper limit to the number of fields we consider for IBFE.
The amount of indicators that are necessary for a given limit \( l \) is \( \sum_{i=0}^{l} \binom{l}{i} \). In Pycket, we chose 5 as the default limit, resulting in 21 pre-defined IBF indicator classes. This seems sufficient, given the average size of Racket structures not overly restrictive, as it covers over 90% of the structure type encountered in the Racket standard library (cf. section 3). Nevertheless, all IBF indicator classes are subjected to the inlining described above, so that each IBF indicator is actually represented by 12 classes for the field inlining.

Hence, when instantiating a structure, Pycket has 252 structure classes to choose from. The operation that maps from all IBF positions to the matching structure class benefits from a lexicographical order of all structure classes; the combination of \#t\ positions determines the position of a structure class uniquely. During instantiation, all positions of immutable fields about to be initialized with \#t\ are shifted to account for their elision. This can also help the inlining optimization, as larger structures with many IBFs now can potentially use an inlined representation instead of a split one.

Accessing an IBF is cheap; with IBFE we make sure that all accesses to those fields are in constant time.

5.4 A Note on Unboxing

As outlined in subsubsection 4.2.3, providing unboxing and mutability with field types is expected to only help for larger structures. It turns out that Pycket already provides limited unboxing capabilities for implementation classes ibf.

1. all fields are immutable,
2. the size is not larger than two, and
3. the stored values are either of Racket’s fixnum or flonum type.

In this case, two or four words of memory can be saved. Given Racket structures are mostly small and mostly immutable. However, we use this automatic unboxing only for the IBFE optimization level.

5.5 Implementation Summary

Overall, the whole structures implementation in Pycket includes 15 implementation classes, about 30 structure primitives, and about 50 general primitives, totalling in about 2000 lines of RPython code.

6. EVALUATION

Pycket is not yet a feature-complete Racket implementation and due to pending (non-structure related) features, the existing Racket structure benchmarks do not run yet. We therefore use a subset of micro-benchmarks instead. We provide an evaluation and execution time and memory consumption based on these benchmarks.

Setup All benchmarks were run on an Intel Core i5 (Haswell) at 1.3 GHz with 3 MB cache and 8 GB of RAM under OS X 10.10.2. All micro-benchmarks are single-threaded. RPython at revision a10c97822d2a was used for translating Pycket. Racket v6.2.0.4 and Pycket at revision 3d0229f were used for benchmarks.

Methodology Every micro-benchmark was run five times uninterrupted. The execution time was measured in-system and, hence, it does not include start-up time. However, it does include warm-up time and the time needed for JIT compilation. We show the execution times of all runs relative to Racket with bootstrapped [10] confidence intervals for a 95% confidence level. The memory consumption was measured as maximum resident set size and is given relative to Racket; the confidence intervals were negligibly small and have been omitted.

6.1 Micro-benchmarks

The micro-benchmark set consists of ten tests. Besides examining basic operations, such as structure creation, call of the predicate procedure and accessing and mutating structure fields, we include two slightly more realistic use-cases.

6.1.1 Basic Operations

We used the following benchmarks for the basic operations: create creates simple structures representing two-dimensional coordinates with integer values; create/super re-uses the create benchmarks, but adds a third dimension using structure type inheritance; create* is the same a create, but with an IBF as first field; create/super* is the same as create/super, but with an IBF as first field; predicate checks the type of given structures including the whole type hierarchy; access performs accesses to various immutable fields of structures; and mutate changes every value of a structure and reads the stored value afterwards on each loop iteration. Each benchmark essentially contains a loop with few basic operations and collects the result in a variable to avoid elimination.

6.1.2 Binary Tree

In the binary tree benchmark, the base structure type represents a leaf, which has only one value. A node is a subtype of the leaf referencing two other nodes. This benchmark tests several operation with structures of multiple types simultaneously. We use two versions of this micro-benchmark, where values of leaves are integers (binarytree) and booleans (binarytree*), respectively.

6.1.3 Parser

The parser benchmark is a Brainfuck interpreter. It creates one instance of a structure referencing a list and a data pointer. The operations on the structure include mutations of the data pointer and accessing list elements, and hence, the parser benchmark tests the structure’s accessor and mutator, but not the constructor. The benchmark’s interpreter executes a simple program that generates a Sierpinsky triangle several times.

6.2 Optimization Impact and Results

We report the impact of all optimizations on execution time and memory consumption. The final performance results of optimized Pycket are shown in Figure 8. Note that we accumulate optimization, as they form dependencies. Hence, for example, inlining includes flat structures. The raw numbers are presented in section 9. By way of example, we show the validity of the predicted memory saving, using the create, create/super, and binarytree benchmarks. For IBFE, we however use their boolean counterparts create*, create/super*, and binarytree*.

6.2.1 Direct Mapping Approach

10Brainfuck is an esoteric programming language that models a Turing machine with eight operations on an array.
In some benchmarks, such as predicate, access, mutate, and also parser, Pycket shows outright better execution time and memory consumption results, even without any optimization (“Pycket”). Expectedly, benchmarks that require the creation of many structures initially show worse performance, for example both create and both binarytree benchmarks.

### 6.2.2 General Optimizations

#### 6.2.2.1 Flat Structure.

This optimization improves performance when the benchmarks frequently create structure instances, for example in all create…and binarytree…benchmarks. The impact on the remaining tests is less pronounced. Some benchmarks with intensive access operations show slightly worse performance results, for example access.

For memory consumption, the flat structure optimization should save $n \cdot (6 \cdot 0.319 + 1)$ words for $n$ structures on average, according to Equation 1. However, in our tests, structures either always have a one super-type or do not have super-types at all. Thus, this formula transforms to $n \cdot (6 + 1) = n \cdot 7$ for structures with exactly one super-type, for example in create/super and binarytree and $n$ for the create micro-benchmark where structures are created without super-type. Having one machine-word equals 64 bit and 15 000 000 structures in the create benchmark results in

$$n = 15000000 \cdot 64 \text{bit} \approx 114.4 \text{MB}.$$ 

The create benchmark shows a benefit of 115 MB. The predicted gain of memory consumption for the create/super benchmark with 30 000 000 structure instances is

$$n \cdot 7 = 30000000 \cdot 7 \cdot 64 \text{bit} \approx 1602.2 \text{MB}$$

which is approximately equal to the result of 1603.8 MB. For binarytree, whose nodes always have one super-type except the leaves, the result for trees of depth 22 is

$$n_1 \cdot 7 + n_2 = (2^{22} \cdot 7 + 2^{23}) \cdot 64 \text{bit} \approx 288 \text{MB}$$

#### 6.2.2.2 Inlining.

As expected, all benchmarks except mutate gained execution time performance, especially for creation heavy benchmarks, where the avoided indirection shows in reduced execution time and memory consumption. The actual memory saving, according to Equation 2, should be $n \cdot 3 \cdot \text{sizeOfSpecializedClasses words}$, for $n$ structure instances. The size of the specialized classes turned out to be insignificantly low. Having 15 000 000 structure instances in the create micro-benchmark, we should save

$$n \cdot 3 = 15000000 \cdot 3 \cdot 64 \text{bit} \approx 343.3 \text{MB}.$$ 

The measurement 344.6 MB differs only slightly. For the create/super micro-benchmark with 30 000 000 instances, we should save

$$n \cdot 3 = 3000000 \cdot 3 \cdot 64 \text{bit} \approx 686.6 \text{MB}.$$ 

Our measurement deviates slightly with 689.8 MB. Finally, for binarytree with a 22-level deep tree, we expected

$$n \cdot 3 = 2^{23} \cdot 3 \cdot 64 \text{bit} \approx 192 \text{MB}$$

which fits our measurement of 192.9 MB. (”+ Inlining”)

#### 6.2.2.3 Mutability separation with cells.

The mutate benchmark achieves a significant speed-up from the cell optimization, as the JIT can now treat the actual structure instance as immutable; the additional indirection pays off. As expected, other performance results remain approximately the same.

The Pycket automatic unboxing for small structures has not been enabled for this optimization level, hence, there is only minor influence of using cells on memory consumption on itself. (”+ Cells”)

---

**Figure 8:** Benchmark results with execution times (left) and memory consumption (right) normalized to Racket. Lower is better.
6.2.3 Immutable Boolean Field Elision

All benchmarks with IBFs— that is create*, create/super* and binarytree*— achieve a speed-up and reduced memory consumption. In these particular benchmarks, the execution time becomes about 30% faster. Memory savings range from 25% to 40%. At the same time, all other benchmarks are virtually untouched, showing next to no disadvantages of employing IBFE. (“+ Booleans opt.”)

The actual memory saving, according to Equation 4, should be
\[
\text{size of SpecializedClasses} \approx \frac{0.35n}{1000000000} \times 26 \cdot 0.26 \cdot 0.7 \cdot 0.85 = \frac{3n}{1000000000} \times 26 \cdot 0.26 \cdot 0.7 \cdot 0.85
\]

The measured result 881.8 MB deviates less than 4%.

6.2.3.1 create*.

Structures have one IBF per (two-field) instance, always being false, all fields immutable, yielding \(n \cdot 2 \cdot \frac{1}{2} = n\). Considering Pycket’s automatic unboxing, create* makes use of Racket’s a fixnum for the second structure field. Hence, compared with the cells optimization level, additional two words per structure are saved, yielding \(n \cdot (2 + 2 \cdot \frac{1}{2}) = 3n\). Having 15 000 000 structure instances for create*, we should save
\[
3n = 3 \cdot 15000000 \cdot 64\text{bit} \approx 343.3\text{MB}
\]

The measured result 344.6 MB differs only slightly.

6.2.3.2 create/super*.

Structures have two IBF per (three-field) instance, always being false, all fields immutable, yielding \(n \cdot 3 \cdot \frac{1}{2} = 2n\). However, the third field being a Racket fixnum, and the number of actual fields dropping from three to one due to IBFE, Pycket’s unboxing applies, and additional two words will be saved per structure instance, eventually yielding \(n \cdot (2 + 3 \cdot \frac{1}{2}) = 4n\). Having 30 000 000 structure instances create/super*, we should save
\[
4n = 4 \cdot 30000000 \cdot 64\text{bit} \approx 915.5\text{MB}
\]

The measured result 881.8 MB deviates less than 4%.

6.2.3.3 binarytree*.

Structures have one IBF with a false per instance, yielding \(n\). With a tree depth of 22, we should save
\[
n = 2^{23}, \text{64bit} = 64\text{MB}
\]

which matches the measured result exactly.

6.3 Limitations

We only evaluated the efficiency of structures in Pycket on self-written benchmarks. Although they are well suited to test performance of basic operations with structures, real-world applications may show different behavior as part of future work. Once feasible, more elaborate benchmarks will be used.

JIT warm-up time has an impact on execution time. We use our benchmarks with a sufficient warm-up time, which is not guaranteed to be always reachable in real-world applications. Also, warm-up time may differ between benchmarks. In order to illustrate the importance of the sufficient warm-up time in micro-benchmarks, we ran the create/super micro-benchmark with different numbers of iterations. The results of this benchmark are presented in Figure 9. Pycket shows pure performance results with a small number of iterations, but starting with some sufficient number (about 30 millions in this particular micro-benchmark). Pycket is continuously faster. The slowness of Pycket at the beginning arise from the JIT warm-up. Therefore, we use different, sufficiently large numbers of iterations in every benchmark to show the well-established performance.

![Figure 9: Execution times (in log(ms)) of create/super micro-benchmarks for Racket and Pycket with different number of iterations illustrate the influence of JIT warm-up. Lower is better.](image)

Finally, we are unable to influence internal CPU optimizations, such as enabling a boost-mode. However, such optimizations should work same for both Racket and Pycket running single threaded.

7. RELATED WORK

Late Data Layout is a lightweight annotations mechanism [22] to eliminate limitations of coercions between internal data representations. Boxing and unboxing operations are not inserted eagerly by a compiler but only at execution time, with checks that ensure the consistency of the data representation. The checks are based on multi-phase type-driven data representation transformations, local-type inference. Hence, unnecessary transformation operations can be omitted and data-type representations are added optimally.

The object storage model [23] of Truffle [24] creates every object as an instance of a storage class, which works as a container for the instance data. This class references a shape that describes the object’s data format and behavior. Shapes and all their accessible data are immutable, but the reference to a shape from the storage class themselves can vary over time. Thus, any change of the object’s shape results in a new shape. The proposed approach is suitable for sufficiently efficient compilation with further optimizations, such as polymorphic inline cache (PIC) for efficient object’s property lookup.

A more specialized approach to increase performance of data structures in VMs is storage strategies [5] for collections of homogeneously typed elements. If possible, they are stored unboxed and their type is stored separately and only once with a special object called strategy. For example, adding an integer to an empty collection enables the integer strategy for this collection and this integer and all subsequent integers will be saved unboxed. However, adding a non-integers, for example a string, causes a transition to a generic
strategy, because the collection is now heterogeneous. It is assumed that such transformations are unlikely, which is shown by the authors. A similar approach is used for structures with mutable cells in this work. Every cell has its strategy and its values are saved unboxed, unless under a generic strategy.

While pointer tagging and strategies reduce memory consumption by unboxing values, it is also possible to reduce the size of the structure itself, when a substantial amount of structures is allocated. Structure vectors group structures of the same type, allowing to store the header and the type descriptor only once [9]. This optimization is most beneficial when large amounts of structures are used, achieving a speed-up of up to 15%. Yet, while allocation becomes faster, field access and especially type descriptor access become up to three to four times slower [9]. However, the allocation of a big number of structures is not very common in Racket (cf. section 3).

An effective run-time representation exists for R6RS Scheme records [16] where each record has an associated execution time representation, record-type descriptor (RTD), determining its memory layout. When an RTD is created, the compiler calculates record sizes and field offsets for this record type similar to the way presented here. They have flat representation with inline fields, quite similar to structures Pycket. A special interface allows to store raw integers, untagged floating point numbers, and raw machine pointers, in addition to ordinary Scheme data types.

The representation of structures in Racket’s implementation is related to our work, too. However, we deliberately chose to not investigate the implementation but rather base our approach solely on the extensive documentation and the static and dynamic analyses. A comparison of our implementation to Racket’s is part of future work.

8. CONCLUSION AND FUTURE WORK

We presented an analysis of record structure usage in Racket and proposed optimizations that are fit for an efficient implementation. We considered three common approaches and devised a novel optimization for immutable boolean fields. We applied these approaches to Pycket, a tracing-JIT-based implementation of Racket, and achieve a significant speed-up compared to Racket in provided micro-benchmarks with a sufficient warm-up time. We evaluated the impact of our optimizations with a set of micro-benchmarks.

Our results suggest further investigation of unboxing values, as homogenised fields in structures make up about 85% in Racket on average. Adaptive optimizations [19] show promising initial results and may be applied to records in the future. Finally, once Pycket’s feature coverage is sufficient, we will run a broader range of benchmarks.

9. APPENDIX: COMPREHENSIVE BENCHMARK RESULTS

The results of all benchmarks are presented in Table 2 (execution time) and Table 3 (memory consumption). The first rows of these tables contain Racket numbers for references. The second row present the unoptimized implementation. All subsequent rows represent improvements with each optimization, in an accumulated fashion, that is, the last row represents Pycket with all optimizations presented here. Benchmarks annotated with * make explicit use of booleans. All error values are bootstrapped [10] confidence intervals for a 95% confidence level.

Acknowledgments

We gratefully acknowledge the financial support of HPI’s Research School and the Hasso Plattner Design Thinking Research Program (HPDTRP). We want to thank Carl Friedrich Bolz for fruitful discussions, Spenser Bauman, Jeremy Siek, and Sam Tobin-Hochstadt for their support during implementation, and the anonymous reviewers for their insightful comments. Carl Friedrich Bolz is supported by the EPSRC Cooler grant EP/K01790X/1.

10. REFERENCES


Table 2: Execution times (in ms) for Racket and Pycket (without optimizations, with flat structures, with inlined fields, with cells, and with **ibfe**). Less is better.

<table>
<thead>
<tr>
<th>VM / Optimization</th>
<th>Create</th>
<th>Create*</th>
<th>Create/sup.</th>
<th>Create/sup.*</th>
<th>Predicate</th>
<th>Access</th>
<th>Mutate</th>
<th>Binary tree</th>
<th>Bin. tree*</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racket</td>
<td>4982 ± 134</td>
<td>5210 ± 114</td>
<td>19,684 ± 726</td>
<td>20,243 ± 97</td>
<td>3585 ± 105</td>
<td>2917 ± 125</td>
<td>4306 ± 223</td>
<td>1817 ± 61</td>
<td>2046 ± 82</td>
<td>1061 ± 52</td>
</tr>
<tr>
<td>Pycket</td>
<td>7027 ± 39</td>
<td>5803 ± 134</td>
<td>35,779 ± 1395</td>
<td>24,657 ± 772</td>
<td>221 ± 8</td>
<td>172 ± 16</td>
<td>1214 ± 14</td>
<td>4735 ± 202</td>
<td>3959 ± 205</td>
<td>715 ± 61</td>
</tr>
<tr>
<td>+ Flat structure</td>
<td>6116 ± 90</td>
<td>5245 ± 140</td>
<td>20,575 ± 742</td>
<td>14,132 ± 160</td>
<td>227 ± 10</td>
<td>162 ± 5</td>
<td>1291 ± 88</td>
<td>3133 ± 126</td>
<td>2379 ± 132</td>
<td>732 ± 54</td>
</tr>
<tr>
<td>+ Inlined fields</td>
<td>4821 ± 169</td>
<td>4002 ± 122</td>
<td>14,682 ± 261</td>
<td>10,654 ± 104</td>
<td>226 ± 11</td>
<td>177 ± 10</td>
<td>1250 ± 23</td>
<td>1976 ± 136</td>
<td>1429 ± 76</td>
<td>667 ± 23</td>
</tr>
<tr>
<td>+ Cells</td>
<td>4886 ± 70</td>
<td>3894 ± 58</td>
<td>14,446 ± 488</td>
<td>10,066 ± 8</td>
<td>224 ± 21</td>
<td>171 ± 6</td>
<td>555 ± 12</td>
<td>1850 ± 115</td>
<td>1504 ± 29</td>
<td>684 ± 53</td>
</tr>
<tr>
<td>+ Booleans</td>
<td>4726 ± 94</td>
<td>2586 ± 77</td>
<td>14,317 ± 432</td>
<td>5517 ± 81</td>
<td>216 ± 14</td>
<td>161 ± 5</td>
<td>387 ± 15</td>
<td>2016 ± 109</td>
<td>1224 ± 51</td>
<td>666 ± 28</td>
</tr>
</tbody>
</table>

Table 3: Memory consumption (in MB) for Racket and Pycket (without optimizations, with flat structures, with inlined fields, with cells, and with **ibfe**). Less is better.

<table>
<thead>
<tr>
<th>VM / Optimization</th>
<th>Create</th>
<th>Create*</th>
<th>Create/sup.</th>
<th>Create/sup.*</th>
<th>Predicate</th>
<th>Access</th>
<th>Mutate</th>
<th>Binary tree</th>
<th>Bin. tree*</th>
<th>Parser</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racket</td>
<td>871.7 ± 0.0</td>
<td>871.7 ± 0.1</td>
<td>1923.9 ± 0.0</td>
<td>1924.0 ± 0.0</td>
<td>50.5 ± 0.7</td>
<td>813.8 ± 0.0</td>
<td>813.8 ± 0.0</td>
<td>576.3 ± 0.7</td>
<td>376.6 ± 0.1</td>
<td>52.1 ± 0.0</td>
</tr>
<tr>
<td>Pycket</td>
<td>1692.5 ± 0.0</td>
<td>1462.5 ± 0.0</td>
<td>5365.4 ± 0.2</td>
<td>4882.6 ± 3.9</td>
<td>6.5 ± 0.0</td>
<td>769.6 ± 0.0</td>
<td>769.7 ± 0.0</td>
<td>875.7 ± 0.1</td>
<td>747.2 ± 0.1</td>
<td>33.3 ± 0.2</td>
</tr>
<tr>
<td>+ Flat structure</td>
<td>1577.4 ± 0.1</td>
<td>1347.6 ± 0.0</td>
<td>3761.1 ± 0.0</td>
<td>2841.6 ± 0.0</td>
<td>6.5 ± 0.0</td>
<td>769.5 ± 0.0</td>
<td>769.7 ± 0.0</td>
<td>587.0 ± 0.1</td>
<td>457.2 ± 0.1</td>
<td>34.7 ± 0.5</td>
</tr>
<tr>
<td>+ Inlined fields</td>
<td>1232.7 ± 0.0</td>
<td>1003.0 ± 0.0</td>
<td>3071.5 ± 0.0</td>
<td>2152.1 ± 0.0</td>
<td>6.5 ± 0.0</td>
<td>769.5 ± 0.0</td>
<td>769.7 ± 0.0</td>
<td>394.1 ± 0.1</td>
<td>265.1 ± 0.1</td>
<td>34.1 ± 0.3</td>
</tr>
<tr>
<td>+ Cells</td>
<td>1232.8 ± 0.0</td>
<td>1003.0 ± 0.0</td>
<td>3071.5 ± 0.0</td>
<td>2152.1 ± 0.0</td>
<td>6.5 ± 0.0</td>
<td>769.5 ± 0.0</td>
<td>769.7 ± 0.0</td>
<td>394.1 ± 0.1</td>
<td>265.1 ± 0.1</td>
<td>34.6 ± 0.3</td>
</tr>
<tr>
<td>+ Booleans</td>
<td>1233.0 ± 0.1</td>
<td>696.1 ± 0.0</td>
<td>3071.8 ± 0.1</td>
<td>1270.3 ± 0.0</td>
<td>6.7 ± 0.0</td>
<td>769.8 ± 0.1</td>
<td>769.9 ± 0.0</td>
<td>394.1 ± 0.1</td>
<td>201.1 ± 0.1</td>
<td>34.9 ± 0.4</td>
</tr>
</tbody>
</table>


ABOUT THE AUTHORS:

Tobias Pape is a Ph.D. student in Robert Hirschfeld’s software architecture group at the Hasso Plattner Institute, University of Potsdam. He’s interested in virtual-machine construction, language design, and data structure optimization. In his spare time, he contributes to the Smalltalk community. Contact him at tobias.pape@hpi.uni-potsdam.de.

Vasily Kirilichev is a Hasso Plattner Institute, University of Potsdam alumnus. His research interests are in software architecture and programming language design, especially optimizations for virtual machines. Contact him at vasilii.kirilichev@hpi-alumni.de.

Carl Friedrich Bolz is a researcher in Computer Science with Laurence Tratt as a member of the Software Development Team at King's College. He received a Ph.D. from the Heinrich-Heine-Universität Düsseldorf, Germany. His main research interests are techniques for the efficient implementation of dynamic programming languages. Also, he is interested in tracing JITs, optimizations, partial evaluation, particularly at runtime. Carl Friedrich is a member of the Python Software Foundation. Contact him at cfbolz@gmx.de.

Robert Hirschfeld is a Professor of Computer Science at the Hasso-Plattner-Institute at the University of Potsdam. There he founded and leads the Software Architecture Group which is concerned with fundamental elements and structures of software, methods, and tools for improving the comprehension and design of complex and interesting systems. He received a Ph.D. in Computer Science from the Technical University of Ilmenau, Germany. Contact him at @hpi. uni-potsdam.de. See also http://www.swa.hpi.uni-potsdam.de/.
AutoFix: An Automated Approach to Memory Leak Fixing on Value-Flow Slices for C Programs

Hua Yan  
University of New South Wales, Australia  
huayan@cse.unsw.edu.au

Yulei Sui  
University of New South Wales, Australia  
y.sui@unsw.edu.au

Shiping Chen  
Data61  
CSIRO, Australia  
shiping.chen@csiro.au

Jingling Xue  
University of New South Wales, Australia  
j.xue@unsw.edu.au

ABSTRACT

C is the most widely used programming language for developing embedded software, operating systems, and device drivers. Unlike programs written in managed languages like Java, C programs rely on explicit memory management, and are therefore prone to memory leaks. Existing (static or dynamic) debugging tools only report leaks, but fixing them often requires considerable manual effort by inspecting a list of reported true and false alarms. How to develop on-demand lightweight techniques for automated leak fixing without introducing new memory errors remains challenging.

In this paper, we introduce AutoFix, a fully automated leak-fixing approach for C programs by combining static and dynamic program analyses. Given a leaky allocation site reported by a static memory leak detector, AutoFix performs a graph reachability analysis to identify leaky paths on the value-flow slices of the program, and then conducts a liveness analysis to locate the program points for inserting fixes (i.e., the missing free calls) on the identified leaky paths. We have implemented AutoFix in LLVM-3.5.0 and evaluated it using five SPEC2000 benchmarks and three open-source applications. Experimental results show that AutoFix can safely fix all the memory leaks reported by a state-of-the-art static memory leak detector with small instrumentation overhead.

CCS Concepts

- Software and its engineering  
- Memory management; Software performance; Software reliability;

Keywords

Memory Leaks; Bug Fixing; Value-Flow Analysis

1. INTRODUCTION

In software testing, the two central tasks facing software engineers are finding bugs and fixing them. Both tasks are expensive in dollar terms and time-consuming due to the ever-increasing scale and complexity of modern software systems. A large number of existing program analyses and testing techniques focus on automatic bug detection. However, finding bugs is only the first step. Once reported, bugs must still be repaired. Indeed, manually fixing bugs can be a non-trivial and error-prone process, especially for large-scale software.

Recently, a few approaches to automatic bug fixing have been proposed to reduce maintenance costs by producing candidate patches for program validation and repairing [1, 11, 15, 22, 24, 37]. For example, CLEARView [24] enforces violated invariants to correct buffer overflow and illegal control flow errors by creating patches for binaries. AutoFixE [37] relies on user specifications and generates repairs using contracts. PACHKA [9] infers object behavior models to propose candidate fixes for bugs like null dereferences. GENPROG [11] uses genetic programming to repair bugs in legacy code.

Most of the existing automatic approaches for fixing bugs in C programs are related to spatial memory errors such as buffer overflows and null pointer dereferences. Such types of bug can be validated by inserting assertions and repaired
Memory leaks represent another major category of memory errors, i.e., temporal memory errors, which are more complicated to fix automatically. Unlike a spatial error that can be fixed by adding a conditional check to bypass the point where the spatial error occurs, every leaky path from a leaky allocation site needs to be traced by inserting an appropriate fix (i.e., a `free` call) without introducing new memory errors.

Figure 1 illustrates how intra- and inter-procedural memory leaks are fixed correctly and incorrectly. Suppose the memory allocated in line 2 in Figure 1(a) is never freed. Adding a fix, `free(p)`, too early in line 5 can cause a use-after-free error in line 11, whereas adding `free(p)` in line 12 at the end of program without considering path correlation may introduce an invalid `free` site for a stack object (when the `if` branch is executed). A correct fix is provided in lines 13–16, with the underlying path correlation accounted for correctly. Let us consider now an inter-procedural leak shown in Figure 1(b), where the memory allocated in line 2 is leaked partially along the `else` branch (lines 8–11). A simple-minded fix, `free(x)`, which is inserted in line 16 in function `use`, is incorrect. Without considering correlated calling contexts, this fix may introduce a double-free in line 6 when the `if` branch is executed. A correct fix is given in line 10, ensuring that only the leak along the `else` branch is fixed.

Existing (static and dynamic) memory leak detectors for C programs only report leaks, but fixing them along every leaky path remains to be done manually by programmers. Dynamic detectors [4, 21] find memory leaks by instrumenting and tracking memory accesses at runtime, incurring high overhead. By testing a program under some inputs, dynamic detectors typically compute an under-approximation which produces no false positives but potentially misses many bugs. In contrast, static detectors [6, 16, 32, 33, 38], which over-approximate runtime behaviors of the program without executing it, can soundly pinpoint all the leaks in the program, but at the expense of some false positives.

This paper presents AutoFix, a fully automated approach to fixing memory leaks in C programs by combining static and dynamic analyses. Given a list of leaky allocation sites reported by a static detector, AutoFix automatically fixes all the reported leaks by inserting appropriate fixes (i.e., `free` calls) along all the leaky paths. There are two main challenges. First, a detector reports a leaky allocation site as long as it discovers one leaky path from the site without necessarily reporting all the leaky paths. AutoFix is designed to fully repair the memory leak for all its leaky paths. Second, some reported leaks are false positives. AutoFix must guarantee memory safety by ensuring that the fixes are correct regardless of whether the reported leak is a true bug or a false positive. Note that AutoFix certainly cannot fix any leaks that are missed (i.e., not reported) by a static detector.

AutoFix applies to a large class of real-world C programs where memory management is explicitly orchestrated by programmers without resorting to garbage collection (GC) and/or reference counting (RC). Compared to the GC and RC approaches, our approach is lightweight as only small instrumentation overhead is incurred. To safely reclaim a leaked memory object `o` from an allocation site without any programmer intervention, all the memory allocation and deallocation sites reachable from `o` on the value-flow slices of the program are instrumented to keep track of the liveness of `o` in shadow memory, thereby enforcing correct leak fixing inside a memory-safe execution sandbox at runtime.

Figure 2 highlights the basic idea behind AutoFix. Given a leaked object `o` from an allocation site (reported by any leak detector), AutoFix builds from the program a sparse value-flow graph (S1), on which a graph reachability analysis is first performed to locate the candidate functions for inserting appropriate fixes, i.e., `free` calls (S2). For each candidate function `f`, AutoFix then identifies the leaky paths in `f` for `o` by computing the value-flow guards with respect to its existing deallocation sites found in the program (S3). Next, a liveness analysis is applied inside `f` on the value-flow slice of the identified leaky paths for `o` to determine every program point `P` where a fix is needed with path correlation considered (S4). Finally, the fixes are inserted immediately after the last use sites of `o` on all its leaky paths (S5). At runtime, the instrumented fixes performs runtime checking to verify statically identified leaks, and reclaims only true leaked memory objects.
This paper makes the following contributions:

- We present AutoFix, a fully automated approach to memory leak fixing for C programs that can reclaim all the leaked memory objects reported by any leak detector.

- We propose an approach to safe memory leak fixing by combining static and dynamic analyses: an inter-procedural static analysis is first performed to identify the earliest program points that the missing free calls can be inserted without introducing any use-after-free error; and then, a dynamic analysis, which operates on efficient shadow memory, tracks the potentially leaked memory objects and performs runtime checking to fix true leaks without introducing any double-free error.

- We have implemented AutoFix in LLVM-3.5.0 and evaluated it using five SPEC2000 benchmarks and three open-source applications. Our experimental results show that AutoFix can safely fix all the leaks reported by the state-of-the-art static leak detector, Saber [32, 33], with runtime overhead averaged at under 2%. For the long-running server application Redis evaluated, AutoFix has successfully reduced its memory usage by more than 300MB in a three-hour continuous run after having fixed its leaks.

2. A MOTIVATING EXAMPLE

In order to describe the main idea of our approach, we use the example in Figure 3 to go through the five key steps shown in Figure 2. An allocation site in line 4 of fun in Figure 3(a) is partially leaky along the else branch in line 10 inside a while loop. Given a leaked object o detected from this allocation site by a static leak detector, AutoFix first constructs a value-flow graph shown in Figure 3(b) for o. Based on this graph, AutoFix inserts one fix immediately after line 16 on the leaky else branch, as shown in Figure 3(c). The leaky allocation site (in line 4) and the deallocation site (in line 8) in the non-leaky if branch are instrumented with dynamic checks to ensure safe fixing at runtime, as illustrated in Figure 3(d).

**Step 1: Constructing Value-Flows.** Following [32, 33], we construct an inter-procedural value-flow graph (VFG) for every leaky allocation, with the one for the example given in Figure 3(b). Note that its nodes are numbered by the corresponding line numbers in Figure 3(a). In a VFG, its nodes represent the definitions of variables and its edges capture their def-use relations.

<table>
<thead>
<tr>
<th>void Fun ( ) {</th>
</tr>
</thead>
<tbody>
<tr>
<td>char* q = ”on stack”;</td>
</tr>
<tr>
<td>while (C0) {</td>
</tr>
<tr>
<td>char* p = malloc(...);</td>
</tr>
<tr>
<td>fgets(p,...);</td>
</tr>
<tr>
<td>if (C1) {</td>
</tr>
<tr>
<td>Use(p);</td>
</tr>
<tr>
<td>free(p);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>else {</td>
</tr>
<tr>
<td>if (C2) {</td>
</tr>
<tr>
<td>char* t = p;</td>
</tr>
<tr>
<td>p = q;</td>
</tr>
<tr>
<td>q = t;</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>Use(p);</td>
</tr>
<tr>
<td>printf(&quot;loop&quot;);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>void Use (char* p) {</td>
</tr>
<tr>
<td>printf(&quot;%s&quot;, p);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Figure 3: A motivating example.
Step 2: Locating Functions to Fix. Given a leaked object \( o \) from an allocation site, AUTOFix first determines the functions where the leaks of \( o \) should be fixed. A candidate function \( f \) is chosen if \( f \) allocates \( o \) (directly in itself or indirectly in its callee functions) such that \( o \) is never returned to any caller of \( f \). Note that the existence of a candidate function for \( o \) is guaranteed since the \texttt{main} function will be the last resort. In our example, \textbf{Fun} is selected as a candidate function since it contains an allocation site of \( o \) (line 4) and \( o \) is never returned to any callers of \textbf{Fun} based on its value-flows.

Step 3: Identifying Leaky Paths. AUTOFix identifies the leaky paths for \( o \) in \textbf{Fun} by reasoning about value-flow guards, which are Boolean formulas capturing branch conditions betweendefs and uses in the control flow graph (CFG). The \texttt{free} condition \( C1 \) under which the \texttt{free} site in line 8 is reached is computed by performing a guarded reachability analysis from the \texttt{malloc} source \( \text{\#} \) to the \texttt{free} sink \( \text{\#} \). Thus, the leak condition \(-C1\) encodes the leaky paths in the \texttt{else} branch as highlighted in red in Figure 3(c).

Step 4: Liveness Analysis. An intra-procedural liveness analysis is performed for \textbf{Fun} to mark the live basic blocks for \( o \) (shown as double-framed boxes in Figure 3(c)) that are reachable from the allocation site of \( o \) on its leaky paths. As a result, node \( F \) is identified as the last-use site of \( o \).

Step 5: Inserting Fixes. As shown in Figure 3(c), a deallocation Fix(\( x \)) is inserted immediately after \( F \) (i.e., line 16), where the last use of \( o \) is found. In addition, the instrumentation code (in dotted-line boxes) also includes the metadata-manipulating functions inserted (after the \texttt{malloc} source \( \text{\#} \) and the \texttt{free} sink \( \text{\#} \)) to maintain runtime shadows for \( o \) to ensure safe fixing for both leaky and non-leaky paths.

Figure 3(d) shows that the shadow memory simply maps an allocation site with its unique ID, \texttt{allocID}, to a hash map that records (start) addresses of the dynamically allocated objects that are not yet freed. Consider Figure 3(c) again. Every address \( p \) that points to an object allocated at \( \text{\#} \) is recorded in the shadow heap by calling function \texttt{Add(allocID, p)} instrumented immediately after \( \text{\#} \). The deallocation site \( \text{\#} \), which is reachable from \( \text{\#} \) via value-flows, is instrumented by calling function \texttt{Remove(allocID, p)} to delete the address \( p \) from the shadow hash map since its pointed-to object has been released along the non-leaky if branch. On reaching Fix(\texttt{allocID}) during program execution, all the objects identified by the addresses corresponding to the allocation site \texttt{allocID} in the shadow memory are freed (as they would be leaked otherwise).

3. AUTOMATED LEAK FIXING

AUTOFix is a compile-time transformation for inserting runtime checks to reclaim leaked memory for C programs, which keeps track of potential leaked memory addresses via an efficient shadow metadata structure. In this section, we first present the five steps of AUTOFix’s compile-time transformation (§3.1 – §3.5), and then describe the design of AUTOFix’s metadata structure (§3.6).

3.1 Step 1: Constructing a Value-Flow Graph

An inter-procedural sparse value-flow graph (VFG) [12, 29, 32, 33, 34, 35, 42, 43] for a program is a multi-edged directed graph that captures the def-use chains of both top-level and address-taken variables. Top-level variables are the variables whose addresses are not taken. The def-use chains for top-level variables are readily available once they have been put in SSA form as is standard. Address-taken variables are accessed indirectly and inexplicitly at loads and stores. Their def-use chains are built in several steps following [7, 32, 33].

![Value-flow graph](image-url)
First, the points-to information for the program is computed by using, e.g., Andersen’s analysis. Second, the accesses of address-taken variables are made explicit by adding annotations. A load \( p = *q \) is annotated with a function \( \mu(x) \) for each variable \( x \) that may be pointed to by \( p \) to represent a potential use of \( x \) at the load; A store \( *p = q \) is annotated with \( x = \chi(x) \) for each variable \( x \) that may be pointed to by \( p \) to represent a potential def and use of \( x \) at the store; A call site \( cs \) is also annotated with \( \mu(x)/x = \chi(x) \) for each variable \( x \) that is inter-procedurally referred/modified inside the callee of \( cs \). A function definition definition \( def(\ldots) \) is annotated at the entry/exit with \( \mu(x)/x = \chi(x) \) for each variable \( x \) that is inter-procedurally referred/modified inside \( f \). Third, all the address-taken variables are converted to SSA form, with each \( \mu(x) \) being treated as a use of \( x \) and each \( x = \chi(x) \) as both a def and use of \( x \). Finally, the value-flows are constructed by connecting the def-uses for each converted SSA variable.

Figure 4 gives an example. The program in Figure 4(a) defines two variables \( a \) and \( b \) in line 1, whose addresses are taken by pointers \( p \) and \( q \), respectively. By indirectly memory accesses via pointers \( p \) and \( q \), \( a \) and \( b \) are first initialized in line 3 and 4 respectively, and are then passed to subroutine \( swap \) in line 5. Finally, \( c \) gets \( a \)'s value ("<" after the swap) by dereferencing \( p \) in line 6, while \( d \) gets \( b \)'s value (">" after the swap) by dereferencing \( q \) in line 7. To track indirect value-flows, annotations are added to make indirect memory accesses explicit, as shown in Figure 4(b), where the subscripts of \( a \) and \( b \) are as in standard SSA form. Note that the subscripts start from zero at each function entry \( (a_0 \) in line 3 for \( \text{main} \) and in line 9 for \( \text{swap} \), \( b_0 \) in line 4 for \( \text{main} \) and in line 9 for \( \text{swap} \)). By connecting def-uses based on the annotations, the value-flow graph in Figure 4(c) is constructed, which shows how \( a \) and \( b \)'s values are exchanged and then flow to \( c \) and \( d \) respectively.

### 3.2 Step 2: Locating Functions to Fix

**Definition 1 (Value-flow Reachability).** A variable \( v \) is \( o \)-reachable if there exists a value-flow path from the allocation site of \( o \) to the definition site of \( v \) on the VFG of the program. A call site \( p = \text{call}(\ldots) \) is \( o \)-reachable if either variable \( p \) or \( x \) in any \( x = \chi(x) \) function annotated at the call site is \( o \)-reachable.

Given a leaked object \( o \) from an allocation site reported by a static detector, AutoFix first determines the candidate functions where the leaks of \( o \) will be fixed. A function \( f \) is a candidate function to insert fixes for \( o \) if (1) \( f \) contains at least one \( o \)-reachable callsite and (2) there is no \( o \)-reachable variable in any caller of \( f \).

In the case of recursion, if there is no data dependence on the leaked object \( o \) between any two function calls in the recursive cycle, \( o \) can be fixed in the recursive functions; otherwise fixes for \( o \) must be put outside the recursive functions to ensure safe fixing. In the case of global variables, since they are reachable for every function, leaked objects pointed to by global pointers can only be fixed in \( \text{main} \). However, global variables are generally not considered as leaks in existing leak detectors [6, 16, 32, 33].

### 3.3 Step 3: Identifying Leaky Paths

To identify the leaky paths in a candidate function \( f \), AutoFix performs a forward analysis on the VFG from an \( o \)-reachable callsite \( src \) to construct a value-flow slice \( S_{src} \) that includes all the nodes reachable from \( src \) but confined in \( f \).

If no \textit{free} sites are reachable from \( src \), then all paths in function \( f \) are leaky paths. Otherwise, for a \textit{free} site \( snk \) corresponding to \( S_{src} \), let \( vfp(src, snk) \) be the set of all value-flow paths from \( src \) to \( snk \) on the VFG, and \( vfe(P) \) be the set of all value-flow edges in a single value-flow path \( P \in vfp(src, snk) \). Thus, we can obtain the value-flow guards from \( src \) to \( snk \):

\[
VFGuard(src, snk) = \bigvee_{P \in vfp(src, snk)} \bigwedge_{(\hat{s}, \hat{d}) \in vfe(P)} CFGuard(\hat{s}, \hat{d})
\]

\[
CFGuard(\hat{s}, \hat{d}) = \bigvee_{Q \in cfp(\hat{s}, \hat{d})} \bigwedge_{e \in boolCond(e)} booleanCond(\hat{s}, \hat{d})
\]

where \( CFGuard(\hat{s}, \hat{d}) \) is a boolean formula that encodes the set of control-flow paths, denoted as \( cfp(\hat{s}, \hat{d}) \), from program point \( \hat{s} \) to \( \hat{d} \) on \( f \)'s CFG. Each branch edge \( e \) on a control flow path \( Q \) is uniquely assigned a boolean variable \( boolCond(e) \). In the presence of loops, guards can grow unboundedly. To avoid unbounded conjunctions that describe all loop iterations, we follow [6, 32, 33] and bound loops to one iteration. Finally, the leak condition for \( src \) is obtained by computing guards from \( src \) to all its reachable \textit{free} sites in \( f \):

\[
\text{LeakCond} = \bigvee_{snk \in S_{src}} VFGuard(src, snk)
\]

Any path from \( src \) to the end of function \( f \) that satisfies \( \text{LeakCond} \) is a leaky path for the leaked object \( o \).

### 3.4 Step 4: Liveness Analysis

For a candidate function \( f \), a subgraph is extracted from \( f \)'s CFG by excluding the control flow edges that are not on any leaky path. Then, a liveness analysis is applied to this
dependence on a leaked object of each block until a fix-point is reached (lines 8 – 11). For variables (lines 6 – 7) and iteratively marking the liveness performed in $f$ as shown in Algorithm 1, a backward data-flow analysis is live.

Based on the liveness information, Algorithm 2 performs a subgraph to determine the basic blocks in $f$ where $o$ may be live.

As shown in Algorithm 1, a backward data-flow analysis is performed in $f$, starting from blocks containing $o$-reachable variables (lines 6 – 7) and iteratively marking the liveness of each block until a fix-point is reached (lines 8 – 11). For a leaked object $o$ allocated inside a loop, if there is no data dependence on $o$ between different loop iterations, then fixes for $o$ can be inserted inside the loop; otherwise fixes for $o$ must be put outside the loop to ensure safe fixing.

3.5 Step 5: Instrumentation

Based on the liveness information, Algorithm 2 performs instrumentation to insert fixes for every leaked object $o$ in each of its candidate functions $f$. A call to Fix() is inserted either at the end of $b$ where $b$ is the last live basic block for $o$ (lines 6 – 7), or at the beginning of $n$ where $n$ is a newly created basic block between a live block and a non-live block of $o$ (lines 9 – 11). As shown earlier in Figure 3(d), these fixes serve to reclaim the dynamically allocated memory at the allocation site of $o$ that would otherwise be leaked. In addition, calls to Add() and Remove() are instrumented to maintain runtime shadow memory, thereby enforcing safe leak-fixing. A call to Add() is inserted after the allocation site of $o$ (line 12) to track all its allocated objects in the shadow memory. A call to Remove() is inserted after each free site reachable from its corresponding allocation site (line 13), so that the freed objects are removed from the shadow memory.

Our instrumentation is safe even if the leaked object $o$ is a false positive, for two reasons. First, the VFG of a program over-approximates its def-use chains. Thus, the last-use sites of $o$ in its candidate functions for fixing $o$ are conservatively found, ensuring safety by avoiding any use-after-free. Second, Add() and Remove() maintain valid memory addresses in the shadow memory, ensuring safety by avoiding any potential double free along any program path.

![Figure 5: The metadata structure design.](Image)

**Algorithm 2: Instrumentation (for leaked object $o$)**

1. Let $F$ be the set of candidate functions for fixing $o$
2. Let $allocID$ be the unique ID of $o$'s allocation site
3. foreach $f \in F$ do
4.  \hspace{1em} Let liveBBs be the set of basic blocks that are marked live for $o$ in $f$'s CFG
5.  \hspace{1em} foreach $b \in liveBBs$ do
6.  \hspace{2em} if $b \in succ(b), \text{isLive}(s) = true$ then
7.  \hspace{3em} Insert Fix($allocID$) at the end of $b$
8.  \hspace{2em} else
9.  \hspace{3em} foreach $s \in succ(b)$ s.t. $\text{isLive}(s) = false$ do
10.  \hspace{4em} Insert a new block $n$ between $b$ and $s$
11.  \hspace{4em} Insert Fix($allocID$) at the beginning of $n$
12. Insert Add($allocID, p$) immediately after the allocation site $p = malloc(...)$ for $o$
13. Insert Remove($allocID, q$) immediately after each free site free($q$) where $q$ is $o$-reachable
3.6 The Metadata Structure

The design philosophy behind our metadata structure is to enable a judicious tradeoff between time and space, which aims to support fast lookup, insertion and removal operations with reasonable space overhead. As shown in Figure 5, AutoFix maintains a closed hash map $H_{allocID}$ for every leaky allocation site (with its unique ID, allocID) to keep track of all the dynamic allocated memory addresses. The size of $H_{allocID}$ can be user-defined, with larger hash maps consuming more space and smaller ones potentially imposing higher slowdown due to hash collisions. To achieve a reasonable tradeoff (as evaluated in §4.3), the default hash map size is set to $2^8$, with 128 bits for each slot, resulting in a total of 4 KB consumed for the hash map.

Without loss of generality, our shadow mechanism is supported on a 64-bit x86_64 architectures with 48-bit virtual address space and word-aligned pointers. For a 64-bit memory address allocated at a leaky site, AutoFix uses its lower 8 bits as the index to the corresponding entry in the hash map, and maps its middle 40 bits to the field ADDR of the entry. A linked list is implemented to handle hash collisions in each hash slot, with the field SC recording the head of the list. Due to the sparsity of the hash map, it is expensive to retrieve all the valid entries to reclaim leaked memory by performing a full scan for the map. To speed up the search, we have used a doubly linked list (similar to Java’s LinkedHashSet) with the two 8-bit fields, PRE and NEXT, to record the previous and the next valid entry indexes. The one-bit field FEN indicates whether an entry is valid or not. It is set to 0 when the entry is removed from the hash map. The 7-bit field USL is preserved for future use.

Figure 6 gives the implementation of our shadow metadata functions i.e. Add(), Remove(), Fix(). The lookup, insert, remove and clear are standard operations that are similar to those in Java’s LinkedHashSet and are thus omitted. Function Add(), which is instrumented immediately after the leaky allocation site allocID, first finds a slot in $H_{allocID}$ to create an entry for the allocated memory object $o$ (line 1), then maps bits 8 through 47 of $o$’s address addr to the 40-bit field ADDR using a simple shift operation (line 2), and finally inserts the entry into $H_{allocID}$ (line 3). Function Remove(), which is instrumented after a deallocation site of $o$, checks whether the deallocated address is recorded in $H_{allocID}$ (lines 4 and 5). If so, the corresponding entry is removed (line 6). Function Fix(), which is instrumented after the last-use site of $o$, first traverses $H_{allocID}$ using its internal linked list via getNext (lines 7), then frees all the recorded addresses (line 8) and clears the hash map (line 9).

4. EVALUATION

We have implemented AutoFix on top of LLVM (version 3.5.0). Eight C programs are used for evaluation as shown in Table 1, including five SPEC2000 benchmarks and three popular open-source applications. The SPEC2000 suite is widely used for evaluating static leak detectors [6, 16, 32, 33]. However, the SPEC2000 benchmarks that have less than two reported leaks (e.g. parser and gap) are excluded from our evaluation. The five selected SPEC2000 benchmarks are ammp (contains many leaks), perlbmkm (allocation-intensive), twolf (allocation-intensive) and mesa (deallocation-intensive). We also include three open-source applications: a2ps-4.14 (a postscript filter) containing a relative large number of leaks, and two long-running server programs: h2o-1.2 (an http server) and redis-2.8 (a NoSQL database).

All our experiments are conducted on a platform consisting of a 3.0 GHZ Intel Core2 Duo processor with 16 GB memory, running RedHat Enterprise Linux 5 (kernel version 2.6.18). The source code of each program is compiled into bit-code files using clang and then merged together using LLVM Gold Plugin at link-time (LTO) to produce a whole-program bit-code file. The compiler option mem2reg is turned on to promote memory into registers. Andersen’s pointer analysis is used to build the VFG for the program [12]. We use the leak warnings (leaky allocation sites) reported by the state-of-the-art leak detector, SABER [32, 33], as input to AutoFix.

We evaluate AutoFix based on three criteria: (1) efficiency (number of fixes generated and the analysis time taken to do so), (2) effectiveness (ability to fix memory leaks and reduce memory usage at runtime), and (3) performance degradation (instrumentation overhead at runtime).

4.1 Efficiency of AutoFix

The compile-time results of AutoFix are summarized in
Table 1: Benchmark characteristics

<table>
<thead>
<tr>
<th>Program</th>
<th>Size (KLOC)</th>
<th>#Alloc Sites</th>
<th>#Free Sites</th>
<th>#Leaky Alloc Sites Reported</th>
<th>#True Leaks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>13.4</td>
<td>37</td>
<td>30</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>gcc</td>
<td>230.4</td>
<td>161</td>
<td>19</td>
<td>45</td>
<td>40</td>
</tr>
<tr>
<td>perlhmk</td>
<td>87.1</td>
<td>148</td>
<td>2</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>mesa</td>
<td>61.3</td>
<td>82</td>
<td>76</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>twolf</td>
<td>20.5</td>
<td>185</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>n2ps</td>
<td>41.8</td>
<td>295</td>
<td>161</td>
<td>39</td>
<td>28</td>
</tr>
<tr>
<td>h2o</td>
<td>18.2</td>
<td>95</td>
<td>123</td>
<td>27</td>
<td>26</td>
</tr>
<tr>
<td>redis</td>
<td>61.8</td>
<td>47</td>
<td>62</td>
<td>24</td>
<td>20</td>
</tr>
<tr>
<td>Total</td>
<td>534.5</td>
<td>1050</td>
<td>474</td>
<td>179</td>
<td>150</td>
</tr>
</tbody>
</table>

Table 2. Given a total of 179 leaky allocations reported in the eight programs, AUTOFIX fixes them all by inserting 393 calls to Fix() (Column 2), 179 calls to Add() (Column 3) and 107 calls to Remove() (Column 4). On average, a leaky allocation results in only 2.2 fixes. This shows that AUTOFIX is able to precisely place fixes along the identified leaky paths with lightweight instrumentation. As shown in Table 2 (Column 5), it takes 216.1 seconds to analyze the 534.5 KLOC for the eight C programs altogether. In particular, AUTOFIX spends 81.7 seconds on gcc, the largest program (230.4 KLOC) studied. The analysis times for the other seven programs are all within one minute.

4.2 Effectiveness of AUTOFIX

To evaluate the effectiveness of AUTOFIX in fixing leaks at runtime, we compare the memory usage of each program before and after automated fixing using VALGRIND [21]. For the five SPEC2000 benchmarks, their reference inputs are used. For the three open-source applications, their own regression test suites are used. For a total of 67 real leaks triggered by the inputs (Column 2 in Table 3), AUTOFIX is able to reclaim all the leaked memory at runtime, which is verified by VALGRIND [21].

In our experiments, we observed that a substantial number of leaks are inter-procedural, involving path correlation. These leaks are ignored and cannot be fixed by the pure static approach LEAKFIX [10] due to the over-approximative nature of static analysis. In contrast, AUTOFIX combines static analysis with runtime checking to enable precise fixing for all leaks including those involving path correlations.

Figure 7 shows a memory leak pattern in gcc and the instrumented code with the leak fixed. The pointer p at the callsite use (in line 6) may point to either (1) a heap object (allocated in line 3) when the if branch is taken, or (2) a stack object (allocated in line 1) otherwise. A leak happens in the former case, while the code is leak-free in the latter case. AUTOFIX tracks the memory allocation by instrumenting an Add after malloc (in line 3) and reclaims only truly leaked memory by performing runtime checks in Fix immediately after the last use site (in line 6) conservatively computed by value-flow analysis.
To further evaluate the effectiveness of AUTOFIX in fixing leaks for long-running programs, we reproduce a real leak which causes memory exhaustion in redis with its corresponding regression tests [20]. As shown in Figure 8, a loop (line 281) is used to query the IPs of slave servers. If a slave server is dead, reconnection attempts are repeatedly made by calling getaddrinfo (line 291), which allocates memory chunks that are never freed, resulting in leaks inside the for loop. As shown in Figure 9, the memory consumption of the leaked version of redis increases around 31.7 KB per second, and over 300 MB are leaked after three hours. AUTOFIX can fix this leak effectively, enabling redis’s memory consumption to remain below 35 MB in the fixed version.

4.3 Runtime Overhead

To measure runtime overhead, each program is executed five times before and after automated fixing respectively, and the average overhead is reported in Table 3. The experimental results show that AUTOFIX only introduces negligible overhead for all the eight programs, 1.06% on average, with the maximum 1.82% observed in a2ps. This confirms that our instrumentation is lightweight, achieved by identifying the required deallocation fixes on the value-flow slices of leaky allocations and tracking leaked objects with simple shadow operations at runtime.

To evaluate the impact of the metadata structure (as described in §3.6) on runtime instrumentation overhead, we choose four different sizes for the hash map used in order to demonstrate the time and space tradeoffs made: 1 (with the hash map degenerating into one linked list), 2<sup>8</sup>, 2<sup>16</sup> and 2<sup>256</sup>. The results are shown in Figure 10.

For the five benchmarks, ammp, gcc, mesa, twolf and redis, the four configurations yield similar overheads. However, for the other three benchmarks, perlmbk, a2ps and h2o, much higher overheads are incurred when their underlying hash maps have degenerated into a single linked list. In this degenerate case, the lookup operations become too expensive, especially when a large number of memory objects are present. When the other three hash map sizes are used, lookup operations can be performed more efficiently. The hash map with 2<sup>16</sup> slots is not very space-consuming, costing 1 KB for each leaky allocation. However, due to its high collision rates, this hash map still results in high overheads for perlmbk, a2ps and h2o. As shown in Figure 10, the hash maps with 2<sup>8</sup> slots (4 KB per leaky allocation) and 2<sup>256</sup> slots (1 MB per leaky allocation) suffer from similar overheads. This indicates that 2<sup>8</sup> slots are already sufficient to guarantee low collision rates, and more slots cannot provide any noticeable performance benefit. For more complicated applications beyond our evaluation, it is still possible that 2<sup>8</sup> slots are not enough to ensure low hash collision rates. In this situation, AUTOFIX allows users to allocate more slots for the shadow hash map to achieve better performance.

To understand how the runtime overhead are caused and distributed, we profile each ADD, REMOVE and FIX operation instrumented. Figure 11 shows the result. For all the programs evaluated, maintaining shadow memory (ADD and REMOVE together) incurs more overhead than reclaiming the leaked memory (FIX) by an average of 31.3%. For the two programs without REMOVE operation instrumented (i.e. ammp and twolf), the ADD operation alone (63% in ammp and 58% in twolf) still accounts for more overhead than the

![Figure 10: Comparing runtime overheads for different metadata structures used in AUTOFIX.](image-url)

**Table 2: Compile-time statistics of AUTOFIX**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Fix()</th>
<th>#Add()</th>
<th>#Remove()</th>
<th>Analysis Time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>20</td>
<td>20</td>
<td>0</td>
<td>0.9</td>
</tr>
<tr>
<td>gcc</td>
<td>74</td>
<td>45</td>
<td>13</td>
<td>81.7</td>
</tr>
<tr>
<td>perlmbk</td>
<td>131</td>
<td>12</td>
<td>28</td>
<td>32.0</td>
</tr>
<tr>
<td>mesa</td>
<td>19</td>
<td>7</td>
<td>9</td>
<td>15.1</td>
</tr>
<tr>
<td>twolf</td>
<td>7</td>
<td>5</td>
<td>0</td>
<td>3.9</td>
</tr>
<tr>
<td>a2ps</td>
<td>51</td>
<td>39</td>
<td>48</td>
<td>17.0</td>
</tr>
<tr>
<td>h2o</td>
<td>61</td>
<td>27</td>
<td>2</td>
<td>9.5</td>
</tr>
<tr>
<td>redis</td>
<td>30</td>
<td>24</td>
<td>7</td>
<td>56.0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>393</strong></td>
<td><strong>179</strong></td>
<td><strong>107</strong></td>
<td><strong>216.1</strong></td>
</tr>
</tbody>
</table>

**Table 3: Run-time statistics of AUTOFIX**

<table>
<thead>
<tr>
<th>Program</th>
<th>#Triggered Leaks</th>
<th>Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>1</td>
<td>1.36</td>
</tr>
<tr>
<td>gcc</td>
<td>13</td>
<td>0.75</td>
</tr>
<tr>
<td>perlmbk</td>
<td>12</td>
<td>0.88</td>
</tr>
<tr>
<td>mesa</td>
<td>3</td>
<td>0.76</td>
</tr>
<tr>
<td>twolf</td>
<td>2</td>
<td>0.89</td>
</tr>
<tr>
<td>a2ps</td>
<td>12</td>
<td>1.82</td>
</tr>
<tr>
<td>h2o</td>
<td>15</td>
<td>1.58</td>
</tr>
<tr>
<td>redis</td>
<td>9</td>
<td>0.66</td>
</tr>
</tbody>
</table>
Fix operation (37% in ammp and 42% in twolf). The Remove operation causes significant proportion of overhead in perlbmk (32%), mesa (38%), and a2ps (43%). These three programs also have relatively high numbers of false memory leaks, as shown in Table 1. AutoFix has to identify and tolerate false memory leaks by the Remove operation when fixing true ones.

5. RELATED WORK

Leak Detection: Memory leak detection has been extensively studied using static [16, 32, 33, 38] or dynamic [13, 21, 27] analysis. Static detectors examine the source code at compile-time without executing the program. SATURN [38] detects memory leaks by solving a Boolean satisfiability problem. SPARROW [16] is based on abstract interpretation, and uses function summaries. FASTCHECK [6] and SABER [32, 33] find memory leaks on the value-flow graph of the program. Dynamic detectors, which find leaks by executing the program, track the memory allocation and deallocation via either binary instrumentation as in VALGRIND [21] or source code instrumentation as in ADDRESSSANITIZER [27].

Leak Tolerance: Another line of research focuses on tolerating leaks at runtime [3, 23, 36]. The basic idea is to delay out-of-memory crashes at runtime by offloading stale objects (regarded as likely leaked) to disks and reclaiming their virtual memory. Upon accessing a mistakenly swapped-out object, the object will be swapped back into the memory, thereby guaranteeing safety. Apart from the space overhead, dynamically detecting stale objects by tracking accesses of memory objects also results in non-negligible time overhead. For example, LeakSurvivor [36] incurs an average runtime overhead of 23.7% even for applications without memory leaks. AUTOFIX, which aims at fixing leaks, is orthogonal to tolerating leaks. Instead of dynamically tracking every memory access of every object to determine objects' liveness, AUTOFIX conservatively approximates the liveness for only leaky objects at compile-time, therefore avoiding high runtime overhead.

Garbage Collection: Garbage collection (GC) can eliminate most memory leaks. However, in type-unsafe languages such as C and C++, it is theoretically impossible to implement sound GC to automatically manage memory. A few unsound (conservative) solutions for C and C++ [2, 14, 25] have been shown empirically to be effective with low space and time overheads, in which memory allocations (e.g. malloc sites) are replaced by special allocators, and memory deallocations (e.g. free sites) are removed from the program, at the expense of the prompt low-cost reclamation provided by explicit memory management. Compared to AUTOFIX’s static fixing on value-flow slices, GC uses runtime object reachability to over-approximate object liveness. In addition, garbage collectors for C and C++ typically need to monitor all static data areas, stacks, registers and heap. In contrast, AUTOFIX only monitors potential leaky allocations reported by leak detectors, which makes AUTOFIX much more lightweight than GC. AUTOFIX and conservative GC can be applied simultaneously, with the former in charge of the leaked memory objects allocated by malloc and the latter in charge of the memory allocated by GC’s special allocators.

Leak Fixing: Memory leaks can be fixed manually or automatically. LEAKPOINT [8] is a dynamic taint analysis that identifies last-use sites of leaked objects by tracking pointers and presents programmers the identified sites as candidate locations for leak fixing. LEAKCHASER [39] relies on user annotations to improve the relevance of bug reports, thereby assisting programmers to diagnose and fix memory leaks. Object ownership profiling has also been applied to assisting manual leak detection and fixing [26]. LEAKFIX [10] is a pure static approach to automatically fixing leaks in C programs. Because it cannot handle false positives produced by other state-of-the-art leak detectors, LEAKFIX relies on its own dedicated leak detector and can fix only some but not all reported leaks. In contrast, our approach combines...
static and dynamic analyses, and is able to automatically fix all the true leaks reported by a detector with small runtime overhead.

Value-Flow Analysis: Value-flow analysis computes inter-procedural def-use information for both top-level and address-taken variables. A prerequisite for value-flow analysis is the pointer/alias information provided by pointer analysis. Value-flow analysis, in turn, can assist pointer analysis to improve precision and scalability [12, 18, 19, 30]. Recently, value-flow analysis has been applied in memory error detection [6, 32, 33, 42], program slicing [28], and inter-procedural SSA analysis [5].

6. DISCUSSION

Like many program analysis and software testing problems, static memory leak detection and fixing are undecidable in general. At its core, it is undecidable whether each leaky path is feasible. Moreover, even if we ignore runtime evaluation of branch conditions and assume all static control-flow paths are feasible, it is still impossible to develop an approach that can statically fix all memory leaks by only inserting missing free calls. This is because it requires the following strict condition to be satisfied for each leaky path $\rho$ of each leaked object $o$: there must exist a program point $l$ on $\rho$ and $l$ must not be on any non-leaky path $\rho'$. If the condition is not satisfied, the free call instrumented may mistakenly free non-leaked memory objects and is therefore unsafe. However, the condition has been proven to be not satisfiable for many memory leaks [40]. As a result, pure static approach (e.g. LEAKFix [10]) can fix only some memory leaks.

The current AUTOFix implementation reclaims leaked memory as early as possible, which minimizes performance degradation caused by memory leaks. However, this is not necessarily the solution with the smallest instrumentation overhead, because postponing a fix may allow multiple instrumentations to be merged, thereby reducing runtime overhead. We leave this as our future work.

7. CONCLUSION

This paper presents AUTOFix, a fully automated approach to memory leak fixing for C programs by combining static and dynamic analysis. Given a leaky allocation reported by a leak detector, AUTOFix performs a graph reachability analysis to identify the leaky paths on the value-flow graph of the program, and then performs a liveness analysis to locate the program points for instrumenting the required fixes on the identified leaky paths at compile-time. To guarantee safe fixing, shadow memory is maintained for the potential leaked memory objects at runtime. Our evaluation shows that AUTOFix is capable of fixing all reported memory leaks with small instrumentation overhead.

8. ACKNOWLEDGEMENTS

This work is supported by ARC grants, DP130101970 and DP150102109.

9. REFERENCES


ABOUT THE AUTHORS:

Hua Yan is a PhD candidate in the School of Computer Science and Engineering at the University of New South Wales, Australia, supervised Prof. Jingling Xue. He also works part-time at CSIRO, Australia, co-supervised by Dr. Shiping Chen. Before that, he was a software engineer in a commercial bank for 3.5 years. Hua Yan received his B.S. and M.S. in computer science from Peking University, China, in 2007 and 2010, respectively. His research interests are software engineering, software testing, software reuse, and program analysis.

Yulei Sui received the Bachelor’s and Master’s degrees in computer science from Northwestern Polytechnical University, Xi’an, China, in 2008 and 2011, and the Ph.D. degree in computer science from the University of New South Wales, Sydney, Australia. He has been a Postdoctoral Fellow in Programming Languages and Compilers Group, University of New South Wales, since 2014. He is broadly interested in the research field of software engineering and programming languages, particularly interested in static and dynamic program analysis for software bug detection and compiler optimizations. He worked as a Research Intern in Program Analysis Group for Memory Safe C project in Oracle Lab Australia in 2013. Dr. Sui was an Australian IPRS scholarship holder, a keynote speaker at EuroLLVM, and a Best Paper Award winner at CGO’13.

Shiping Chen is a principal research scientist at CSIRO Australia. He also holds an adjunct associate professor title with the University of Sydney through teaching and supervising PhD/Master students. He has been working on distributed systems for over 20 years with focus on performance and security. He has published over 100 research papers in these research areas. He is actively involved in computing research community through publications, journal editorships and conference PC services, including WWW, EDOC, ICSOC and IEEE ICWS/SCC/CLOUD. His current research interests include secure data storage & sharing and secure multi-party collaboration. He is a senior member of the IEEE.

Jingling Xue received the BSc and MSc degrees in computer science and engineering from Tsinghua University in 1984 and 1987, respectively, and the PhD degree in computer science and engineering from Edinburgh University in 1992. He is currently a professor in the School of Computer Science and Engineering, UNSW Australia, where he heads the Programming Languages and Compilers Group. His main research interest has been programming languages and compilers for about 25 years. He is currently supervising a group of postdocs and PhD students on a number of topics including programming and compiler techniques for multi-core processors and embedded systems, concurrent programming models, and program analysis for detecting bugs and security vulnerabilities. He has served in various capacities on the Program Committees of many conferences in his field. He is the senior member of the IEEE.