Erkennung und Analyse von Speicheranomalien in Sprachen mit automatischer Speicherverwaltung unter Nutzung von Trace-basierter Speicherüberwachung[1]

Markus Weninger[2]

Abstract: Moderne Programmiersprachen nutzen automatische Speicherbereinigung, um fehleranfällige manuelle Speicherverwaltung zu vermeiden. Dennoch können Anomalien wie Speicherlecks auftreten, die sich drastisch auf die Leistung einer Anwendung auswirken und sogar Abstürze herbeiführen können. Die meisten modernen Werkzeuge nutzen für ihre Speicheranalysen jedoch leider nur Speicherauszüge, d.h. sie inspizieren den Speicher nur an einem oder wenigen bestimmten Zeitpunkten. Diese bieten aber oft nicht genug Details, um zur Ursache des Problems vorzudringen. Unser Ansatz nutzt daher Traces, kontinuierliche Aufnahmen von Ereignissen wie beispielsweise Allokationen oder Speicherbereinigungsoperationen. Diese Arbeit zeigt, wie Traces genutzt werden können, um die (automatische) Speicherproblemerkennung und -analyse zu verbessern. Sie schlägt unter anderem Algorithmen zur Aufzeichnungsverarbeitung vor und führt neuartige Anomalieanalysen (z.B. die automatisierte Analyse des Wachstums von Datenstrukturen) sowie interaktive Visualisierungstechniken ein. Ferner untersucht sie, wie (unerfahrene) Benutzer sich bei der Speicheranalyse verhalten und wie Werkzeuge verbessert werden können, um diese Nutzer besser zu unterstützen und anzuleiten.

1 Motivation

In systemnahen Sprachen wie beispielsweise C liegt es in der Verantwortung des Programmierers, Allokationen und Deallokationen korrekt vorzunehmen. Während dies Flexibilität bietet und hochperformanten Code ermöglicht, kann diese manuelle Speicherverwaltung leicht zu unbeabsichtigten Speicherfehlern führen. Die häufigsten dieser Fehler sind auf Speicherlecks (vergessene `free()` Operationen) oder hängende Zeiger (Zugriffe auf bereits freigegebenen / nicht allokierten Speicher) zurückzuführen. Am besten kann dies wohl mit dem Mantra *Aus großer Kraft folgt große Verantwortung* zusammengefasst werden.

Um diesen Problemen entgegenzuwirken, nutzen automatisch speicherbereinigende Sprachen wie beispielsweise Java *Garbage Collection (GC)*, zu deutsch (lit.) *Müllabfuhr*. Während einer solchen werden Objekte, die nicht länger (indirekt) von Speicherwurzeln (engl. *GC roots*, wie beispielsweise statische Felder oder lokale Variablen) aus erreichbar sind, automatisch entfernt und ihr Speicher freigegeben. Leider ist die automatische Speicherbereinigung ebenso anfällig für mögliche Speicherprobleme. Diese können die

[1] Englischer Titel der Dissertation: Detection and Analysis of Memory Anomalies in Managed Languages Using Trace-Based Memory Monitoring

[2] Johannes Kepler Universität Linz, Institut für Systemsoftware, Altenbergerstraße 69, 4040 Linz, Österreich markus.weninger@jku.at

Applikation verlangsamen, falls Entwickler achtlos mit Objektallokationen und der Speicherung von Objekten umgehen. Im schlimmsten Fall können Speicherlecks dazu führen, dass die Applikation nicht nur langsamer wird, sondern komplett abstürzt.

Speicherlecks in Sprachen mit GC treten auf wenn Objekte, die nicht mehr benötigt werden, durch Programmierfehler weiterhin von Speicherwurzeln aus erreichbar bleiben [MBR10]. Beispielsweise kann ein Programmierer vergessen, Objekte aus langlebigen Datenstrukturen zu entfernen, obwohl sie nicht mehr benötigt werden. Der Garbage Collector kann diese Objekte dann nicht freigeben – sie häufen sich also an und der Speicher läuft voll [XR13].

Eine weitere Speicheranomalie ist *hohe Speicherfluktation* (engl. *high memory churn*, auch *excessive dynamic allocations* [PH16; SW00] oder *high allocation density* [Du03]). Dieses Problem tritt auf, wenn Objekte (unnötigerweise) in hoher Frequenz allokiert werden, nur um sie kurz nach ihrer Erzeugung wieder freizugeben. Dies führt dazu, dass Laufzeit sowohl für die Allokation der Objekte selbst, als auch für die Speicherbereinigung ebendieser durch den Garbage Collector, aufgewendet werden muss. Beide Punkte beeinflussen die Performanz einer Applikation negativ.

Um dem entgegenzuwirken, ist es daher von entscheidender Bedeutung, Entwicklern intelligente Speicherwerkzeuge an die Hand zu geben, welche (semi-automatische) Analysen und Funktionen zum Aufspüren, Verstehen und Beheben von Speicheranomalien bieten.

2 Überblick

Da Speicherauszüge (engl. *heap dumps*) oft nicht genug Details bieten, um zur grundsätzlichen Ursache eines Speicherproblems vorzudringen, fokussiert sich diese Arbeit auf die Nutzung von Speicheraufzeichnungen (engl. *memory traces*) zur Anomalieerkennung und -analyse. In ihren Anfängen hatten Traces das Problem, dass der Aufwand, der betrieben werden musste, um diese aufzuzeichnen, oft die Laufzeit der analysierten Applikation um mehr als das 100-fache erhöhte [He06; RGM13; Xu13]. Dies machte Traces irrelevant für Einsätze in Realumgebungen. In den letzten Jahren wurde jedoch gezeigt, dass die Laufzeiterhöhung, welche durch die Aufzeichnung verursacht wird, auf wenige Prozent reduziert werden kann [LBM15; LBM16; Le16]. Ein Großteil der bestehenden Arbeiten in diesem Bereich (1) fokussiert sich entweder auf die (effiziente) Sammlung von Speicheraufzeichnungen, ohne jedoch neuartige Analysen basierend auf diesen Daten zu zeigen, oder (2) präsentiert stark problemspezifische Traceformate, welche für eine bestimmte Art der Speicheranalyse genutzt werden können, untersuchen aber nicht, ob diese auch für weitere Arten der Speicheranalyse genutzt werden könnten.

Diese Thesis beschäftigt sich aus diesem Grund mit der Nutzung von generellen Speicheraufzeichnungen (d.h., Speicheraufzeichnungen die sich nicht auf eine bestimmte Analyse fokussieren). Als Beispiel dienen jene Traces, die durch die AntTracks VM, einer virtuellen Maschine von Lengauer et al. [LBM15; LBM16; Le16] basierend auf der Java Hotspot

VM [Or21], produziert werden. Im Speziellen untersucht die Arbeit die Frage, wie Traces für Analysen in entwicklerorientierten Speicheranalysewerkzeugen genutzt werden können. Um dies zu untersuchen, fokussieren wir uns auf Fragen und Herausforderungen in Bezug auf die *Verarbeitung und Aufbereitung* von Traces, wie man basierend auf ihnen eine gemeinsame Datenbasis für verschiedenste Analysen und Visualisierungen schaffen kann, und wie solche Analysen und Visualisierungen aussehen können, um Entwickler bestmöglich beim *Erkennen, Analysieren und Beheben* von Speicheranomalien zu unterstützen. Weiters untersucht die Arbeit, wie sich (unerfahrene) Benutzer während der Speicheranalyse verhalten, auf welche Probleme und Hindernisse diese dabei stoßen und wie Speicherwerkzeuge im Allgemeinen verbessert werden können, um diese Nutzer besser zu leiten.

3 Themengebiete

Die Ergebnisse dieser Dissertation wurden in 13 Publikationen veröffentlicht, zwei davon in referierten Fachzeitschriften, die anderen auf internationalen Konferenzen. Wir verzichten auf Grund von Längenbeschränkungen auf Einzelreferenzen und verweisen auf die Gesamtdissertation [We21]. Die Arbeiten können in folgende Themen unterteilt werden.

3.1 Speicheraufzeichnungen und deren Verarbeitung

Wir präsentieren einen neuartigen Ansatz zur Speicherdatenaufbereitung basierend auf einem flexiblen Klassifikationssystem. In seinem Kern gruppiert das System Heapobjekte nach vom Nutzer bestimmten Eigenschaften. Diese Eigenschaften können beispielsweise der Objekttyp, die Allokationsstelle, der allokierende Thread, oder ein beliebiges anderes Nutzer-spezifiziertes Merkmal sein. Gruppiert man Heapobjekte nach mehreren Gesichtspunkten, indem man beispielsweise alle Objekte zuerst nach ihren Typen gruppiert und anschließend alle Objekte eines Typs nach deren Allokationsstellen, so resultiert dies in einem *Speicherbaum* (siehe Abb. 1). Wir zeigen, dass solche Speicherbäume als Datenquelle für diverse Analysen wie beispielsweise der Heapanalyse, der Datenstrukturwachstumsanalyse, als auch für verschiedenste Visualisierungen geeignet sind. Unsere Arbeit zur Heapobjektgruppierung wurde auf der *ICPE 2018* für den *Best Paper Award* nominiert.

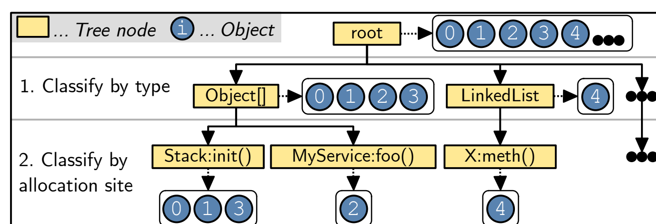
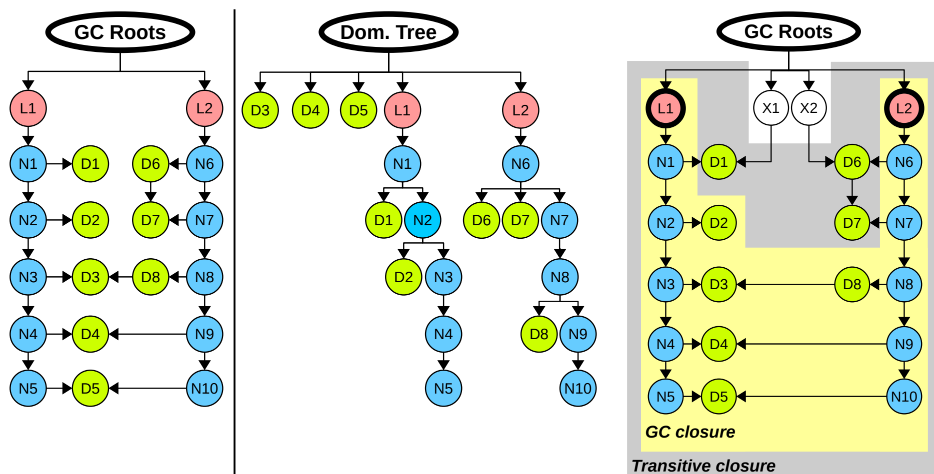


Abb. 1: Ein Speicherbaum, der zuerst alle Objekte nach deren Typ und anschließend nach deren Allokationsstelle gruppiert.

Die Thesis behandelt außerdem die Analyse von Objektreferenzen, d.h., welche Objekte welche anderen Objekte am Leben halten. Die meisten existierenden Ansätze greifen dazu auf die *Dominanzrelation* und *Dominanzbäume* zurück. Der gravierendste Nachteil dieser Ansätze ist jedoch, dass auf Basis der Dominanzrelation nur *Einzelobjektbesitz* analysiert werden kann, d.h., es werden nur jene Objekte erkannt, die durch *genau ein* anderes Objekt am Leben gehalten werden (engl. *single-object ownership*). Dies wird in Abb. 2a veranschaulicht, in dem zu sehen ist, dass für drei Datenobjekte `D3`, `D4` und `D5` mit Hilfe der Dominanzrelation kein Besitzer gefunden werden kann, da sie sowohl von der Liste `L1` als auch von `L2` am Leben gehalten werden. Unser Ansatz unterscheidet sich von Dominanzrelation-basierten Ansätzen darin, dass es uns möglich ist, Objekte und deren Besitzerverhältnis zu erkennen, auch wenn diese von *mehreren* anderen Objekten am Leben gehalten werden (engl. *multi-object ownership*). Dazu präsentieren wir Algorithmen zur Berechnung der *transitiven Hülle*, d.h., alle Objekte, die ausgehend von einem Objekt, oder aber auch einer Gruppe an Objekten, erreichbar sind, sowie Algorithmen zur Berechnung der *Garbage Collection Hülle*, d.h., jene Objekte, die ebenfalls vom Garbage Collector bereinigt werden können, sollte das Ausgangsobjekt, oder eine Gruppe an Ausgangsobjekten, zur Garbage Collection freigegeben werden. Abb. 2b zeigt beide Hüllenarten ausgehend von den beiden verketteten Listen `T1` und `T2`. Hier ist zu sehen, dass unsere Hüllenalgorithmen `D3`, `D4` und `D5` als Teil der Hüllen von `L1` und `L2` erkennt, da diese von beiden Listen am Leben gehalten werden. Solche Hüllen bieten wichtige Metriken zur Erkennung von verdächtigen Speichermustern, und unser Ansatz unterstützt die Analyse dieser Speichermuster somit sogar im Falle von multi-object ownership.



(a) Objektgraph von zwei einfach verketteten Listen `L1` und `L2` sowie deren Dominanzbaum.

(b) Transitive Hülle sowie Garbage Collection Hülle von $L1$ und $L2$.

Abb. 2: Objektgraph, Dominanzbaum, sowie die Hüllen zweier einfach verketteter Listen *L1* und *L2*.

3.2 Datenstrukturanalyse

Die zuvor erwähnten Hüllenalgorithmen sind besonders nützlich zur Analyse von Datenstrukturen und deren Entwicklung über Zeit, denn sie erlauben es uns, *Datenstrukturwachstum* zu erkennen. Wir entwickelten eine domänenspezifische Sprache, um zu beschreiben, welche Teile einer Datenstruktur *intern* sind (wie beispielsweise die Knoten einer `HashMap`) und welche Teile *extern* (wie beispielsweise Objekte, die als Schlüssel oder Werte in einer `HashMap` gespeichert werden). Kombiniert man diese Beschreibungen mit dem Wissen über die zeitliche Entwicklung der Hüllen jeder Datenstruktur, können Metriken abgeleitet werden, die es uns ermöglichen, Datenstrukturen basierend auf ihrer Wahrscheinlichkeit, an einem Speicherleck beteiligt zu sein, zu sortieren. Darüber hinaus können wir die Art von Lecks analysieren: ob es sich um ein datenstrukturinternes oder -externes Leck handelt und ob die anhäufenden Objekte von einem einzelnen Objekt am Leben gehalten werden (single-object ownership) oder von mehreren (multi-object ownership). Der gesamte Analyseprozess ist in Abb. 3 zusammengefasst.

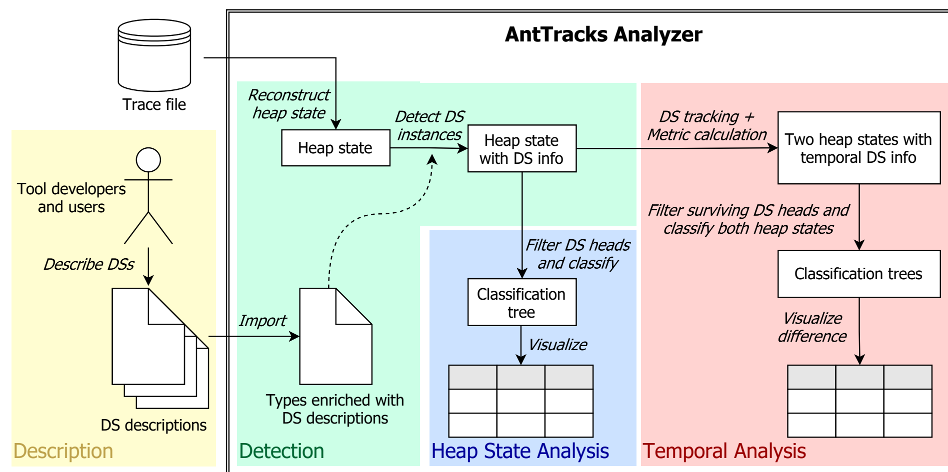


Abb. 3: Datenstrukturanalyseansatz aus 4 Stufen: (1) Beschreibung der Datenstrukturen mit unserer DSL, (2) Erkennung von Datenstrukturen in rekonsturierten Heapzuständen, (3) Heapanalyse, d.h., Datenstrukturanalyse zu einem bestimmten Zeitpunkt, und (4) Wachstumsanalyse, d.h., Erkennung verdächtig wachsender Datenstrukturen durch deren Verfolgung über Zeit.

3.3 Visualisierungen

Weiters präsentieren wir verschiedene Visualisierungen, die die Speicherentwicklung *über Zeit* darstellen. Dafür untersuchten wir bestehende Visualisierungsansätze [Sc11] und evaluierten sie in Bezug auf deren Anwendbarkeit und Nützlichkeit zur Darstellung von Speichermetriken, speziell der Entwicklung von Speicherbäumen über Zeit. Wir zeigen Anwendungen

und Adaptionen von traditionellen Zeitseriendiagrammen, zweidimensionalen *Baumvisualisierungen* sowie einer interaktiven dreidimensionalen *Speicherstadt*-Visualisierung. All diese Ansätze stellen neuartige Inspektions- und Interaktionsmechanismen zur Verfügung, die es erlauben, die Speicherentwicklung eines System leichter verständlich und greifbarer zu machen. Für jeden Ansatz präsentieren wir eine komplette Visualisierungpipeline [Li19], die Datenaufbereitung, Layouting, Darstellung, sowie mögliche Nutzerinteraktionen umfasst.

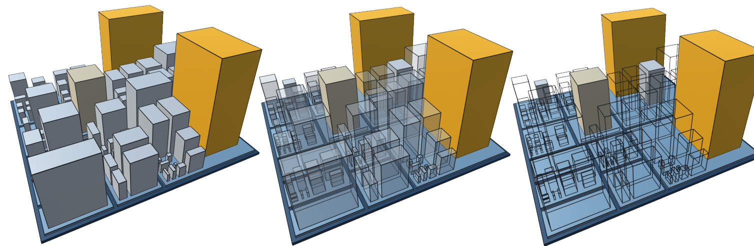


Abb. 4: Unsere Softwarestädte haben vielseitig modifizierbare Darstellungsmerkmale. Links: Jedes Gebäude ist voll deckend dargestellt. Mitte: Die fünf am stärksten wachsenden Gebäude sind voll deckend dargestellt, der Rest hat eine Deckkraft von 40%. Rechts: Die fünf am stärksten wachsenden Gebäude sind voll deckend dargestellt, die restlichen voll transparent (bis auf ihre Umrisse).

Beispielsweise adaptieren unsere Speicherstädte (Abb. 4) die Metapher der *Softwarestadt* [WL07]. Diese wurde in der Vergangenheit meist dazu genutzt, statische Metriken eines Softwaresystems (Klassenhierarchien, etc.) zu visualisieren. Wir erweitern diesen Ansatz um die Darstellung des *dynamischen Speicherverhaltens* einer Applikation. Gruppen von Heapobjekten (beispielsweise Objekte mit dem selben Objekttyp die in der selben Methode allokiert wurden) werden als Gebäude dargestellt, welche wiederum in Distrikten (beispielsweise alle Gebäude des selben Objekttyps) angeordnet sind. Die Größe eines Gebäudes entspricht dabei der Anzahl an Heapobjekten die es repräsentiert. Indem wir die Stadt kontinuierlich aktualisieren (entweder manuell durch den Nutzer oder durch das Abspielen einer automatischen Animation), erzeugen wir das Gefühl einer sich entwickelnden Stadt, in der wachsende Gebäude wachsende Objektgruppen im Heap darstellen. Durch die Nutzung von Farbe und Deckkraft lenken wir die Aufmerksamkeit der Nutzer noch weiter auf bestimmte Gebäude (zum Beispiel jene mit starkem Wachstum), mit denen dann interagiert werden kann, um diese genauer zu untersuchen.

Beide Arbeiten der VISSOFT 2020 (3D Softwarestadt, siehe Abb. 4) sowie der STAG 2020 (2D Baumvisualisierung, siehe Abb. 5) wurden mit dem *Best Paper Award* ausgezeichnet.

3.4 Speicherfluktationsanalyse

Wir haben im Zuge dieser Thesis nicht nur Speicherlecks, die wohl häufigsten Speicheranomalien, untersucht, sondern auch andere Speicherprobleme. Beispielsweise präsentieren wir eine Technik, die Nutzer automatisch auf Zeitfenster mit intensiver Speicherfluktation

hinweist. In solchen Zeitfenstern wird in kurzer Zeit eine große Menge an Objekten allokiert und kurz darauf wieder freigegeben. Wir zeigen, wie man für jedes Heapobjekt seinen *Geburtszeitpunkt* und *Bereinigungszeitpunkt* aus Speichertraces rekonstruieren kann, um daraus die *Lebensspanne* der Objekte zu berechnen, d.h., wie viele Garbage Collections jedes Objekt überlebt hat, bevor es vom GC wieder freigegeben wurde. Wir nutzen diese Informationen, um den Nutzer zu problematischen Stellen im Code zu leiten, um dort häufig allokierte Objekte genauer auf ihre Notwendigkeit hin zu untersuchen.

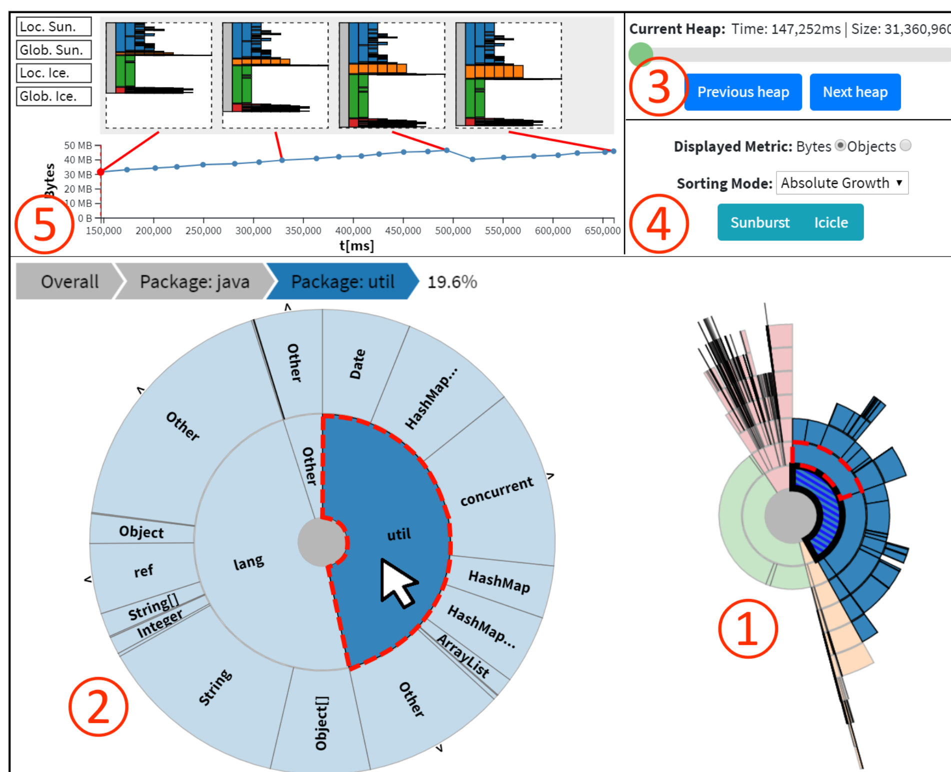


Abb. 5: Übersicht über unser Speicherbaum-Visualisierungstool. (1) und (2) zeigen eine *Zeitreisen-basierte* Visualisierung, wobei (1) den kompletten Baum mit allen Zweigen und Leveln zu einem bestimmten Zeitpunkt zeigt und (2) einen Teilausschnitt dieses Baumes zeigt, dessen Wurzel durch Klicken auf ein Baumsegment gewechselt werden kann. Beide Visualisierungen sind synchronisiert. Beispielsweise ist das Baumsegment, über dem sich der Mauszeiger befindet, sowohl in (1) als auch in (2) hervorgehoben (rote Umrandung). (3) zeigt die Oberfläche zur Zeitkontrolle und (4) bietet verschiedene Visualisierungsoptionen. Daneben befindet sich die (5) *Zeitlinien-basierte* Visualisierung, welche mehrere Bäume an verschiedenen Zeitpunkten nebeneinander anzeigt.

3.5 Anwenderunterstützung und Nutzerverhalten

Wir haben eine Nutzerstudie sowie strukturierte kognitive Durchgänge (engl. *structured cognitive walkthroughs* [BG03]) durchgeführt, um den Nutzen unserer Techniken zu überprüfen sowie ein besseres Verständnis dafür zu erlangen, wie unerfahrene Benutzer Speicherwerkzeuge und deren Ananlysemöglichkeiten nutzen. Basierend auf den Resultaten haben wir Empfehlungen ausgearbeitet, welche Speicherwerkzeugentwicklern helfen sollen, bestehende System zu verbessern und neue Funktionen zu implementieren.

Diesen Vorschlägen folgend haben wir selbst ein verbessertes Nutzerleitsystem in unsere Speicheranalysetechniken eingebaut. Wir entwickelten Algorithmen, die es ermöglichen, automatisch Muster in der Speicherauslastung einer Applikation zu erkennen, welche auf Speicheranomalien wie Speicherlecks oder Speicherfluktationen hinweisen. Wenn ein Zeitfenster mit einem solchen Muster erkannt wird, wird es hervorgehoben, um die Aufmerksamkeit des Nutzers darauf zu lenken.

Basierend auf dieser ersten Erfahrung haben wir ein Konzept zur Nutzerunterstützung namens *Geleitete Exploration (GE)* (engl. *guided exploration*) entworfen. Dieses Konzept dreht sich um vier Leitoperationen, die von Werkzeugen zur Verfügung gestellt werden sollen (siehe Abb. 6): (1) Automatische *Erkennung* von verdächtigen Mustern, (2) *Hervorheben* relevanter Nutzeroberflächenelemente, (3) *Erklärungen* über das beobachtete Muster und warum dieses unerwünscht ist sowie (4) *Vorschläge* von möglichen nächsten Schritten. Unsere Implementierung von GE im AntTracks Analyzer Werkzeug bietet diese Operationen auf jedem Schritt durch die Speicherleckanalyse sowie durch die Speicherfluktationsanalyse, um so Nutzer bestmöglich durch den kompletten Analyseprozess zu leiten.

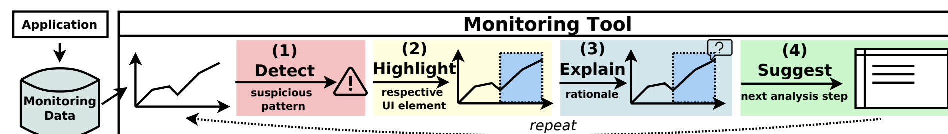


Abb. 6: Die vier Schritte der *Geleiteten Exploration*: (1) Erkennung, (2) Hervorheben, (3) Erklärungen, und (4) Vorschläge.

4 Zusammengefasste Kontributionen

Diese Thesis präsentiert (1) Algorithmen und Datenstrukturen zum Nutzbarmachen und Aggregieren von Speichertraces, (2) einen semi-automatischen Ansatz zum Erkennen und Inspizieren von verdächtig wachsenden Datenstrukturen, (3) verschiedene Visualisierungstechniken (basierend auf Zeitseriendiagrammen, 2D Baumvisualisierungen und der 3D Softwarestadt-Metapher), um die Speicherentwicklung einer Applikation zugänglicher und greifbarer zu machen, (4) eine Technik zum Erkennen und Inspizieren von hoher Speicherfluktation, (5) eine Nutzerstudie und strukturierte kognitive Durchgänge zur Evaluierung des Nutzens unserer präsentierten Techniken, sowie generelle Empfehlungen für Entwickler

von Speicherwerkzeugen, welche final zu (6) unserem Konzept der Nutzerunterstützung *Geleitete Exploration* führten, das speziell unerfahrene Benutzer durch Speicheranalysen leiten soll.

Ein Ziel dieser Arbeit war hervorzuheben, wie flexibel Speicheraufzeichnungen sein können. Wir zeigen interessante Anwendungsfälle, in denen diese eingesetzt werden, um Entwickler dabei zu unterstützen, Speicheranomalien aufzuspüren, zu untersuchen und zu beheben. Darüber hinaus zeigen wir, dass zeitliche Information, d.h., Information über die Entwicklung des Heaps über Zeit, von äußerstem Nutzen sein kann, um detaillierte Speicheranalysen wie beispielsweise Datenstrukturwachstums- und trendanalysen durchzuführen. Solche Analysen über Zeit sind nur mit Speichertraces möglich, da herkömmliche Speicherauszüge (heap dumps) es nicht erlauben, die Entwicklung von Objekten über einen Zeitraum zu beobachten. Auch wenn die Erzeugung von Speichertraces derzeit noch nicht weit verbreitet ist, so hoffen wir, dass Forschung wie diese als Motivation dienen kann, diesen Umstand zu ändern.

Neben theoretischen Konzepten resultierte die Thesis in einer Vielzahl von technischen Kontributionen: die Verbesserung der *AntTracks VM*, die Entwicklung des *AntTracks Analyzer* Werkzeugs, unserem 3D-Visualisierungswerkzeug *Memory Cities*[3], sowie unserem Web-basiertem 2D-Visualisierungswerkzeug *WebTreeViz*[4], welche alle öffentlich verfügbar sind. Wir nutzten diese Werkzeuge als Machbarkeitsnachweise für alle präsentierten Ideen, um deren Anwendbarkeit und Nutzen zu unterstreichen.

Literatur

[BG03] Blackwell, A.; Green, T.: Notational Systems — The Cognitive Dimensions of Notations Framework. In: HCI Models, Theories, and Frameworks. Interactive Technologies, 2003.

[Du03] Dufour, B.; Driesen, K.; Hendren, L.; Verbrugge, C.: Dynamic Metrics for Java. In: OOPSLA. 2003.

[He06] Hertz, M.; Blackburn, S. M.; Moss, J. E. B.; McKinley, K. S.; Stefanović, D.: Generating Object Lifetime Traces with Merlin. ACM Trans. Program. Lang. Syst. 28/3, Mai 2006.

[LBM15] Lengauer, P.; Bitto, V.; Mössenböck, H.: Accurate and Efficient Object Tracing for Java Applications. In: ICPE. 2015.

[LBM16] Lengauer, P.; Bitto, V.; Mössenböck, H.: Efficient and Viable Handling of Large Object Traces. In: ICPE. 2016.

[Le16] Lengauer, P.; Bitto, V.; Fitzek, S.; Weninger, M.; Mössenböck, H.: Efficient Memory Traces with Full Pointer Information. In: PPPJ. 2016.

[3] Video: `http://ssw.jku.at/General/Staff/Weninger/AntTracks/VISSOFT20/MemoryCities.mp4`

[4] Werkzeug:`http://bit.ly/STAG-MemoryTreeVizTool`; Video: `http://bit.ly/STAG-MemoryTreeVizVideo`

[Li19] Limberger, D.; Scheibel, W.; Döllner, J.; Trapp, M.: Advanced Visual Metaphors and Techniques for Software Maps. In: VINCI. 2019.

[MBR10] Maxwell, E. K.; Back, G.; Ramakrishnan, N.: Diagnosing Memory Leaks using Graph Mining on Heap Dumps. In: SIGKDD. 2010.

[Or21] Oracle: The HotSpot Group, last visited on 2022-02-03, 2021.

[PH16] Peiris, M.; Hill, J. H.: Automatically Detecting Excessive Dynamic Memory Allocations Software Performance Anti-Pattern. In: ICPE. 2016.

[RGM13] Ricci, N. P.; Guyer, S. Z.; Moss, J. E. B.: Elephant Tracks: Portable Production of Complete and Precise GG Traces. In: ISMM. 2013.

[Sc11] Schulz, H.-J.: Treevis.net: A Tree Visualization Reference. IEEE Computer Graphics and Applications 31/6, S. 11–15, 2011.

[SW00] Smith, C. U.; Williams, L. G.: Software Performance Antipatterns. In: WOSP. 2000.

[We21] Weninger, M.: Erkennung und Analyse von Speicheranomalien in Sprachen mit automatischer Speicherverwaltung unter Nutzung von Trace-basierter Speicherüberwachung, Dissertation, Institut für Systemsoftware / Johannes Kepler Universität Linz, 2021.

[WL07] Wettel, R.; Lanza, M.: Visualizing Software Systems as Cities. In: VISSOFT. S. 92–99, 2007.

[XR13] Xu, G.; Rountev, A.: Precise Memory Leak Detection for Java Software Using Container Profiling. ACM Trans. Softw. Eng. Methodol. 22/3, 2013.

[Xu13] Xu, G.: Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In: OOPSLA. 2013.

Markus Weninger wurde am 4. April 1992 geboren. Seine Informatikausbildung begann er an der Höheren Technischen Bundeslehranstalt Leonding für *EDV & Organisation*. Danach absolvierte er das Bachelorstudium *Informatik* und das Masterstudium *Computer Science* mit Schwerpunkt *Software Engineering* an der Johannes Kepler Universität (JKU) Linz. Während seines Masterstudiums sammelte er bereits erste Forschungserfahrung am *Institut für Systemsoftware*, wo er sich mit dem Aufbereiten von Speicherüberwachungsdaten beschäftigte. Aus dieser Forschung ging anschließend seine Dissertation zum Thema Trace-basierte Speicheranalyse hervor, welche er im Juli 2021 mit ausgezeichnetem Erfolg verteidigte. Weiters wurde seine Dissertation mit dem *Award of Excellence* des österreichischen Bundesministerium für Bildung, Wissenschaft und Forschung ausgezeichnet. Derzeit arbeitet Markus als Senior Lecturer im Bereich Computer Science an der JKU und vereint dort im Unterrichten von Fächern wie *Softwareentwicklung*, *Compilerbau* oder *Algorithmen und Datenstrukturen* seine Liebe zur Informatik und zur Lehre.