УДК 004.7:004.415-025.25 **doi: 10.26906/SUNZ.2023.2.080**

М. О. Волк, М. В. Гора, В. Г. Лабазов, А. В. Міщенко, А. І. Барсуков, В. В. Голець

Харківський національний університет радіоелектроніки, Харків, Україна

ЖУРНАЛІЗАЦІЯ СТАНУ ПРОГРАМ ДЛЯ САМОВІДНОВЛЕННЯ ПАРАЛЕЛЬНИХ ПРОГРАМНИХ СИСТЕМ

Анотація. У статті розглянута система самовідновлення паралельного програмного забезпечення з використанням журналізації точок відновлення. Самовідновлення є необхідною властивістю сучасних програмних засобів, яка надає можливість автоматичного виявлення, діагностики та відновлення працездатності систем. Основними етапами відновлення є зберігання стану програм (журналізація) в точках відновлення, моніторинг стану програм для виявлення помилок, створення патчів, відновлення стану програм до відповідної точки відновлення. У роботі запропоновано структуру системи, описано алгоритм її функціонування; обговорюються питання призначення та віртуалізація точок відновлення; наведено опис експериментальної програмної системи та її застосування для відновлення поширених програмних систем. Результати можуть набути чималого поширення та застосовуватись у розробці більшості програмних, інформаційних систем з метою автоматизації відновлення, налагодження та експлуатації сучасних комп'ютерних та хмарних систем.

Ключові слова: програмна система, самовідновлення, точка відновлення, стан програми, точка відновлення.

Вступ

Самовідновлення є необхідною властивістю сучасних інформаційних систем, що надає здатність автоматично виявляти, діагностувати та вирішувати проблеми, які виникають у системі, без втручання людини. Ці системи розроблені таким чином, щоб бути відмовостійкими, тобто вони здатні продовжувати функціонувати, навіть якщо певні компоненти відмовляють. Окрім автоматичного вирішення проблем, системи самовідновлення також можуть вживати профілактичних заходів, щоб уникнути можливих проблем у майбутньому. Це може включати передбачення потенційних проблем на основі шаблонів або тенденцій у системних даних і вжиття профілактичних заходів для запобігання виникненню цих проблем.

Інформаційні системи, що самовідновлюються, стають все більш важливими, оскільки підприємства та організації все більше покладаються на технології для роботи. Зводячи до мінімуму час простою та зменшуючи потребу у втручанні людини у вирішення системних проблем, системи самовідновлення можуть підвищити доступність, надійність і продуктивність системи.

Системи самовідновлення використовують різні методи, такі як резервування, реплікація та моніторинг, щоб гарантувати, що система залишається в робочому стані та може швидко відновлюватися після будь-яких збоїв.

Наприклад, якщо сервер у системі самовідновлення виходить з ладу, система може автоматично перенаправляти трафік на інші сервери в мережі, щоб користувачі не постраждали.

В роботі [1] було наведено класифікаційну базу щодо відмов, процесів зберігання стану компонентів системи та задач забезпечення самовідновлення комп'ютерних систем. Зокрема сформульовані основні складові вирішення проблеми самовідновлення: створення механізму самовідновлення, моніторинг стану системи (діагностика та прогнозування збоїв), виявлення порушення нормальної роботи, корегування стану системи, створення умов для запобігання повторення помилки.

Стосовно самовідновлення програмних засобів, як складової інформаційних систем, одним з ефективних методів самовідновлення є створення бекапів – точок зберігання стану програм в фіксований момент часу.

Аналіз існуючих підходів щодо самовідновлення програмних систем

Існує багато підходів для відновлення програмного забезпечення після збоїв. Найбільш поширені з них використовують методи^

перезавантаження, включаючи перезапуск усієї програми [2],

оновлення програмного забезпечення [3] і мікроперезавантаження [4], які повертають систему до начального стану до або після виявлення несправності.

Повний перезапуск програми може зайняти багато часу, що призведе до значного простою програмної системи.

Мікроперезавантаження може бути швидшим, якщо перезапустити лише частину програм системи, але вимагає повного аналізу стану програм для розуміння причин невдачі. Жоден із цих методів неефективно справляється з детермінованими помилками, оскільки вони можуть повторюватися після перезапуску.

Методи відкату до контрольної точки [5] можна використовувати у спосіб, подібний до перезапуску всієї програми, але може забезпечити швидший час перезапуску, оскільки перезапуски виконуються з контрольної точки. При такому використанні ці методи все ще не обробляють детерміновані помилки, оскільки ці помилки все одно виникатимуть після перезапуску.

Також були запропоновані інші варіанти використання контрольної точки-перезапуску в поєднанні з запуском кількох версій програми [6], які можуть пережити детерміновані помилки, якщо збої відбуваються незалежно. Однак вони несуть непо-

мірні витрати для більшості програм з точки зору розробки, підтримки та запуску кількох версій програми одночасно.

Автоматична генерація сигнатур для систем виявлення вторгнень у мережу [7, 8] захищає від вразливостей, фільтруючи вхідні дані для відсівання атак. Ключова проблема полягає в тому, що такі сигнатури досить сприйнятливі до помилкових спрацьовувань, особливо для поліморфних атак.

Крім того, було показано, що поліморфна поведінка надто різноманітна, щоб її можна було ефективно моделювати за допомогою сигнатур [9].

У проекті Vigilante [10] покращено мережеву фільтрацію вхідних даних завдяки автоматичному створенню вхідних фільтрів на базі хоста. Фільтри на основі хоста пропонують покращену точність для виявлення семантично еквівалентних вхідних даних. На жаль, вони потребують специфічних для протоколу парсерів і не можуть працювати зі складними правилами, шифруванням і конкретним станом програми.

Проект Rx [11] використовує механізм перезапуску контрольної точки в поєднанні з механізмами для зміни середовища виконання з метою відновлення після помилок.

Однак попередні роботи [12] виявили, що понад 86% помилок програм не залежать від операційного середовища, є повністю детермінованими та повторюваними, і що відновлення, ймовірно, буде успішним лише за допомогою методів, пов’язаних із програмою.

У той час, як Rx розглядає ширші можливості для зміни середовища, включаючи відкидання шкідливих запитів на введення, на практиці виявилося неефективним через поліморфну поведінку [13]. Rx намагається замаскувати прояв несправностей для клієнта, але потребує використання проксі-сервера додатка з підтримкою протоколу, який має бути здатний відфільтрувати інформацію, таку як мітки часу, яка могла б заплутати клієнтську програму.

Використання проксі-сервера ускладнює використання все більшої кількості програм, які використовують шифрування. Rx вимагає змін ядра операційної системи, що є ще однією перешкодою для розгортання. Нарешті, Rx не вирішує проблеми узгодженості під час встановлення контрольних точок і перезапуску програм, що включають кілька процесів.

"Прибиральник" (sweeper) [14] поєднує механізм перезапуску контрольної точки Rx і проксі з VSEF. Якщо виникає помилка, Sweeper використовує аналіз "забруднення", щоб визначити вхідні дані, які призвели до збою, генерує вхідний фільтр для видалення цього та подібних майбутніх запитів, а потім повертається до попередньої контрольної точки та повторно відтворює вхідні дані. Оскільки Sweeper зменшує VSEF до використання для генерації вхідного підпису, він страждає від тих самих обмежень фільтрації введення, які описані раніше (поліморфізм і зашифрований трафік).

Обчислення, орієнтовані на "прийнятність" (acceptability) [15] просуває ідею про те, що поточні зусилля з розробки програмного забезпечення можуть бути неправильно спрямованими, ґрунтуючись на спостереженні про те, що можна знехтувати певними регіонами програми без негативного впливу на загальну доступність системи.

Обчислення без збоїв [16] — це спекулятивна техніка відновлення, яка базується на компіляторі і вимагає вставку додаткового коду для роботи із записами в нерозподілену пам’ять шляхом віртуального розширення цільового буфера. Така можливість має на меті забезпечити більш надійну реакцію на помилку, ніж просто збій, хоча й із значними накладними витратами на продуктивність, які варіюються від 80% до 500% для різноманітних програм.

Вибіркова трансакційна емуляція (STEM), яка використовується в реактивній імунній системі [17] – це спекулятивна техніка відновлення, розроблена двома авторами, яка визначає функцію, у якій сталася помилка, а потім вибірково емулює цю функцію та, можливо, інші в більшому діапазоні, щоб повернути значення помилок у спробі відновлення після помилки. STEM використовує поняття віртуалізації помилок, щоб означати евристичне значення помилки від функції, у якій сталася помилка. Це дуже відрізняється від поняття віртуалізації помилок точки відновлення, яке використовується в системе, яке повторно використовує існуючий код обробки помилок у програмах і повертає значення на основі профілювання цих функцій для імітації поведінки системи в умовах контрольованих і очікуваних помилок. На відміну від STEM, наші пропозиції не потребують вихідного коду, працюють з багатопроцесорними та багатопоточними програмами, забезпечують суттєві покращення продуктивності системи.

Найбільш формалізований підхід до відновлення розподіленого програмного забезпечення присутній у роботах [18, 19], де на прикладі програмних моделей зорганізується управління станом програми у часі на основі дампів пам'яті. Якщо використати цей підхід до нашої задачі, можлива організація самовідновлення на низькому рівні (операційної системи, віртуальної машини тощо).

Тож **метою** статті є опис механізму самовідновлення паралельного програмного забезпечення з використанням бекапів стану програм на основі журналізації точок відновлення.

Забезпечення життєвого циклу програмних систем з самовідновленням

Система забезпечує архітектурну підтримку для самовідновлення програми за наявності непередбачених збоїв у повністю автоматичному режимі. Система постійно стежить за програмою на наявність збоїв і визначає стратегії, використовуючи точки відновлення для реагування на майбутні випадки таких самих або подібних збоїв.

Після вибору стратегії, система динамічно модифікує програму, використовуючи динамічну бінарну ін’єкцію, щоб вона могла виявити та оминути ту саму помилку в майбутньому. Метою нашої системи є автоматичне створення тимчасового ви-

правлення певної проблеми, доки не стане доступним рішення користувача.

Рис. 1 ілюструє роботу системи на високому рівні. Перед розгортанням, програма профілюється для виявлення потенційних точок відновлення. Після завершення профілювання, програма розгортається у своєму робочому середовищі.

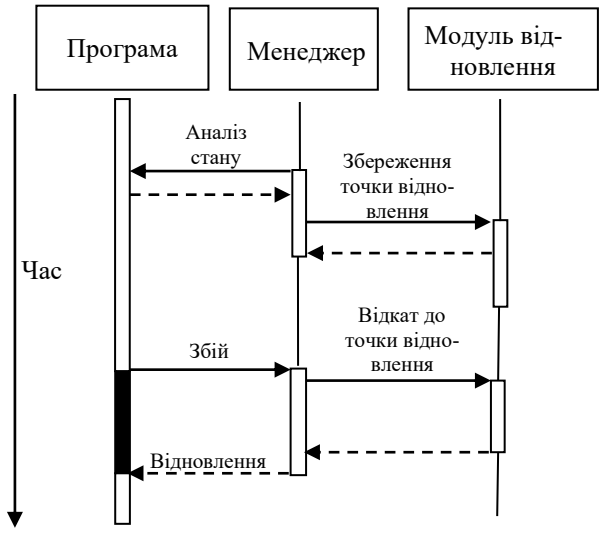

Рис. 1. Журналізація точки відновлення та відновлення системи після збою

Під час нормального виконання, система контролює програму за допомогою інструментальних засобів, які виявляють та звітують про неправильну поведінку програми та системи.

Крім того, система періодично перевіряє стан програми та веде журнал виконання (включаючи мережевий трафік), де періодично фіксує контрольну точку зі збереженням стану програмної системи.

Коли, під час виконання, виявляється помилка, стан останньої контрольної точки програми разом із журналом усіх вхідних даних, починаючи з цієї контрольної точки, передається в систему сортування, де аналізується помилка.

Потім система виконує автоматичний процес, метою якого є визначення відповідної точки відновлення, до якої програма може відновити виконання, при умові, що ця конкретна помилка не виявиться повторно. Протягом цього часу робоча система залишається вразливою до повторного виникнення помилки, що призводить до вікна вразливості, у якому програмі може знадобитися вдатися до повного перезапуску програми для відновлення нормального процесу виконання. Хоча нашій системі може знадобитися деякий час простою для фази аналізу, досвід показує, що це зазвичай декілька секунд, і вартість амортизується, оскільки вона виникає один раз за нову помилку.

Після вибору точки відновлення, система підтверджує, що вона придатна для розгортання, перевіряючи, що вона задовольняє трьом критеріям: живучість, правильність і продуктивність. Вибрана точка відновлення забезпечує живучість, якщо віртуалізація помилок у цей момент дозволяє програмі вижити після повторення помилки.

Точка відновлення є дійсною, якщо вона не вносить семантичних помилок і якщо програма може правильно обслуговувати майбутні запити. Точка відновлення є справжньою, якщо продуктивність захисту не спричиняє значних витрат часу виконання.

Живучість перевіряється шляхом відтворення послідовності подій, які, очевидно, спричинили несправність. Правильність перевіряється за допомогою ретельного тестування, яке адаптовано для конкретної роботи програми.

Ефективність оптимізується за рахунок використання продуктивності як метрики при виборі точки відновлення. Щойно відповідну точку відновлення буде перевірено, система створює виправлення, яке динамічно застосовується до програмного забезпечення, поки програма виконується в робочій системі.

Патч створює точку відновлення всередині програми, щоб захистити програму від повторення конкретної помилки. Змінена програма запускатиме точку відновлення кожного разу, коли виконання досягне точки відновлення, і повертатиме свій стан до цієї точки, якщо помилка повториться.

Після відкату виконання до точки відновлення, віртуалізація помилок використовується для використання існуючих можливостей обробки помилок програми для ефективного усунення помилок.

Замість фільтрації певних вхідних даних, які можуть спричинити збої, патч захищає програму від збоїв, які можуть виникнути в певному місці програми.

Отриманий механізм відновлення не залежить від вхідних даних і, отже, захищений від ризиків, пов’язаних із поліморфізмом помилок/вхідних даних.

Структура системи з самовідновленням

Система забезпечує загальний механізм, який програми можуть використовувати для відновлення виконання за наявності помилок. Запропонований підхід можна розділити на наступні етапи офлайнових і онлайнових дій.

В автономному режимі додатки профілюються під час «хибних запусків», щоб побудувати модель поведінки додатків. Ці хибні прогони генеруються регресійними тестами, якщо вони доступні, або за допомогою методів фаззингу [3,10], який підкреслює можливості обробки помилок програм. Дії полягають в тому, що існує набір тестованих програмістами точок програми, які регулярно використовуються для поширення «очікуваних» помилок.

У свою чергу, ці прикладні точки можна використовувати для відновлення після збоїв і таким чином підтримувати доступність системи.

Використовуючи цю модель, ми виділяємо місця розташування програми, які можна використовувати як потенційні точки відновлення. Наша архітектура в режимі он-лайн дозволяє використовувати різноманітні монітори несправностей. Після виявлення помилки, стан програми повертається до по-

передньо визначеного (точки відновлення), де програма змушена повертати помилку, імітуючи поведінку, що спостерігається під час помилкових запусків. Модулі системи, представлені на рис. 2:

набір засобів (зондів), який постійно відстежує програму на наявність помилок і бере на себе контроль щоразу, коли вони виявляються;

помилка віртуалізації активує компонент, який відповідає за визначення значень для введення в разі несправності;

виявлення точки відновлення активує компонент, який використовується для визначення потенційних точок відновлення за допомогою статики та аналізу;

активатор, який використовує механізм перезапуску контрольної точки для фіксації стану програми та відкату до збереженого стану;

генератор патчів, який створює патчі з підтримкою точки відновлення для аплікації;

середовище тестування, в якому запропоновані патчі оцінюються згодом;

вставка засобів, які полегшують інсталяцію затверджених патчів у запущені бінарні файли.

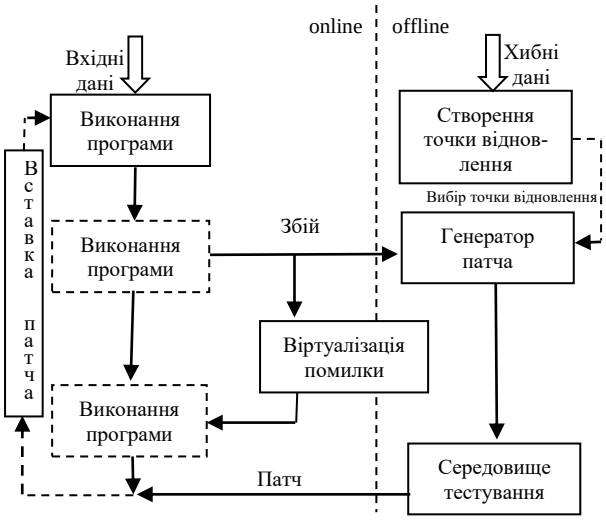

Рис. 2. Робота системи з самовідновленням

Усі компоненти розроблені для роботи в автоматичному режимі, щоб мінімізувати час реакції.

Призначення та віртуалізація точок відновлення

Визначення прийнятної точки відновлення має ключове значення для віртуалізації помилок. Це багато в чому визначає ступінь, ймовірність того, що програма продовжить роботу при несправності. Для виявлення відновлення використовуються два механізми: один статичний і один динамічний.

Динамічний аналіз є кращим механізмом для виявлення відповідних точок відновлення, оскільки він надає однозначне розуміння поведінки програми. Зокрема, метою є дізнатися, як реагує програма на «хибні вхідні дані» у контрольованих умовах і використовувати це знання, щоб зробити так, щоб раніше невидимі дефекти бути прогнозованими. Існує набір перевірених програмістами точок програм, які є регулярними та використовуються для обробки «очікуваних» помилок. Наприклад, можна побачити, як зазвичай поширюється програма помилки під час стрес-тестування тестами забезпечення якості.

Навчання на основі помилок використовувалося в машинному навчанні і згодом системи виявлення помилок, було використано для відновлення програмного забезпечення при помилках. Зокрема, ми інструментуємо програми шляхом вставки коду моніторингу в точках входу та виходу кожної функції, які можливі впровадження під час виконання [2]. Засоби записують обидва параметри функції - типи та значення, що повертаються програмою при помилці. З цих записів можна побудувати графіки викликів функцій разом із інформацією про тип повернення для кожної точки на графіку ("графіки відновлення"). Rescue-графи використовуються як програмні фрагменти на функціональному рівні деталізації, які, у свою чергу, можна використовувати щоб ізолювати контрольний потік несправностей та визначити можливе відновлення.

Статичний аналіз використовується для посилення результатів описаних методів динамічного аналізу. Зокрема, ми використовуємо статичний аналіз для полегшення віртуалізації помилок і виявлення точки відновлення. Для віртуалізації помилок статичний аналіз може допомогти визначити відповідні значення повернення помилок через перевірку коду та зворотні фрагментації програми. Детально досліджуються шляхи, що виникають при прояві несправності, де вразлива функція знаходиться під час виклику. На цьому етапі перевіряється, як значення функції, що повертається, використовується в коді обробки. Це забезпечує розуміння того, як ми можемо використовувати під час помилки механізм віртуалізації.

Наприклад, коли функція повертає значення оператора керування, за яким слідують оператори виходу – це створює умови для використання цього значення як відповідного відновленого значення під час віртуалізації помилок. Протягом цього процесу ми також приділяємо увагу фрагментам програми для будь-яких проблемних випадків віртуалізації помилок, включаючи введення-виведення, яке виконується разом із використанням глобальних змінних та існування коду обробки сигналів.

Опишемо основну концепцію віртуалізації помилок за допомогою точок відновлення. Віртуалізацію помилок можна зробити такими кроками:

зберегти стан програми в точці відновлення;

контролювати систему на наявність помилок;

коли виникає помилка, скасувати зміни стану, зроблені функцією, аж до точки відновлення;

після відновлення виконання до точки відновлення.

Компоненти виявлення несправностей можна розглядати як чорний ящик, якому потрібно лише повідомити про виникнення несправності. На додаток до стандартної обробки помилок операційної системи, можна використовувати додаткові механізми для виявлення помилок пам'яті.

Існує ряд доступних компонентів виявлення несправностей, які можуть виявляти помилки пам'яті (наприклад, ProPolice [9] і TaintCheck [12]) і деякі, які виявляють порушення основних політик безпеки [1,11].

Проведення експериментів з прототипами системи самовідновлення

Експериментальну симуляцію роботи системи було реалізовано для операційної системи Linux. Було використано стандартні утиліти та модулі ядра ядра Linux, які забезпечують віртуальне середовище виконання з точками перезапуску та відтворення журналу, а також Dyninst 10.1 для аналізу паралельного коду під час виконання.

Використовуючи цей прототип, було оцінено ефективність системи щодо реальних помилок і стандартних робочих навантажень для низки популярних багатопроцесних і багатопоточних серверних програм. Для всіх експериментів процес був повністю автоматизованим, за винятком генерування інформації профілю та ініціювання помилки. Процес профілювання має відбуватися один раз для кожної програми (або надаватися як частина тестового пакету). Усі експерименти проводилися на машинах з процесорами Intel Core i7, 8/16, 2,9-4,8 ГГц і 16 ГБ оперативної пам'яті, з підключенням через з'єднання Ethernet 1 Гбіт/с.

Сервери та клієнти працювали на окремих машинах. Оцінка функціонування системи у вирішенні помилок відбувалася за трьома напрямками: живучість, правильність і продуктивність. Живучість перевіряла здатність системи підтримувати доступність служби за наявності збою програмного забезпечення, спричиненого помилкою.

Система виявляла збої та автоматично починала процес відновлення. Після відновлення, сервер перевіряв збої, які могли бути спричинені нашим механізмом, та продовжував правильно обслуговувати запити. Оскільки можливо, що механізм відновлення вносить побічні ефекти, перевірка на правильність вихідних даних сервера проходила і після відновлення.

Потім проводили оцінку продуктивності системи з точки зору як повних системних накладних витрат на робочому сервері, так і часткової перевірки компонентів системи.

Було використано п'ять помилок для трьох популярних програм: Apache (версія 2.2), MySQL (8.0), і PostgreSQL (12.14). Серед помилок були: порушення прав доступу до пам'яті, переповнення буфера, некоректні вхідні дані. Також, для порівняння ми закладали зовнішні помилки для Linux. Дослідження виконано для програм із відкритим вихідним кодом.

Помилки генерувалися під час виконання тесту, паралельно з цим програма відстежувалась, щоб перевірити її здатність успішно пройти контрольний тест. Якщо тест завершувався, виконувалась перевірка живучості та оцінка продуктивності. Програма перевірялася на правильність або шляхом перевірки результатів порівняльного тесту, або через додаткові тести, які перевіряли і порівнювали вихідні дані з очікуваним набором результатів. Для кожної помилки фіксувалася глибина відновлення та значення відновлення: відстань між несправністю і точкою відновлення та значення віртуалізації помилки, яке потім використовувалась для поширення помилок.

Результати часової оцінки різних етапів процесу підтримки відновлення представлено на рис. 3.

Рис. 3. Оцінка часу виконання операцій при виявленні помилки

Було реалізовано наступні помилки:

(1) переповнення буферу (Apache);

(2) невірні вхідні дані (Apache);

(3) порушення прав доступу до даних (MySQL);

(4) переповнення буферу (MySQL);

(5) невірні вхідні дані (PostgreSQL).

Щоб оцінити швидкість реагування системи на створення виправлення для нововиявленої помилки, вимірювався загальний час, необхідний для переходу від помилки до виправлення. Іншими словами, від моменту першого виявлення несправності у виробничій системі до динамічного застосування виправлення. Однак, треба відмітити, що такі процеси, як моніторинг та створення точки відновлення може виконуватись паралельно з роботою системи. Створення патча для зафіксованої помилки виконується один раз та зберігається у системі. Таким чином, коли помилка трапляється наступного разу, система виконує тільки відновлення.

Рис. 3 показує середній час у секундах для створення робочого, перевіреного патча для кожної з помилок. Вказується загальний час, необхідний для створення, тестування та застосування патча. Загальний час розбивається на дві частини: час, необхідний для підключення до запущеного процесу та введення точки відновлення за допомогою інструментів системи, і тестовий час, необхідний для виконання тесту живучості, правильності та продуктивності в середньому для успішної точки відновлення. Зауважемо, що час, необхідний для створення графіка відновлення для конкретної помилки, незначний, враховуючи, що траса стека була глибиною менше 5 функцій у всіх випадках.

Для кожної помилки ми оцінювали ефективність відновлення після несправності. Зокрема, час відновлення стану програми до точки відновлення після виявлення несправності. Помилка виникала, коли програма була зайнята виконанням заданих тестів для вимірювання відновлення під навантаженням.

Ми порівняли час відновлення системи із часом перезапуску всієї програми після збою, у якому ми виміряли час, що минув від запуску програми до моменту, коли вона стане працездатною та готовою обслуговувати запити. Повний перезапуск програми не обов’язково дозволяє відновлення, але це може скинути сервер, щоб він міг обслуговувати майбутні запити, навіть якщо він не дозволяє завершити робоче навантаження, яке виконувалося на момент помилки. Хоча він не забезпечує такого ж рівня живучості, як наша система, він забезпечує відновлення. Як видно з рис. 3, перезавантаження призводить к значним витратам часу в порівнянні з етапами запропонованого механізму самовідновлення. Щоб виміряти реалістичний час перезапуску додатка, ми тестували перезапуск програми за допомогою реальних навантажень. Наприклад, для PostgreSQL ми вимірювали час, необхідний для перезапуску програми, коли в ній попередньо завантажено набір даних Wisconsin, для Apache визначили середній час, необхідний для відновлення виконання до точки відновлення, або, іншими словами, відкат до точки колишнього виконання.

Висновки

Розроблена система надає можливість створення нового програмного забезпечення для самовідновлення програмних систем, зокрема для відновлення від програмних збоїв у серверних програмах. Точки відновлення – це місця, які визначені в існуючому коді програми, де обробка помилки виконується щодо заданого набору передбачених або виявлених збоїв.

Існуючі методи перевірки достовірності можна використовувати для створення відомих хибних вхідних даних програм, щоб визначити потенційні точки відновлення.

При першому виявленні несправності, система використовує репліку (стан програми у точці відновлення) програми, щоб визначити, які точки відновлення можна використовувати найбільш ефективно для відновлення майбутнього виконання програми.

Після того, як система перевірить, чи створено виправлення, яке усуває несправність, вона динамічно виконує віднолення. Якщо помилка виникає знову, система повертає програму до точки відновлення та використовує власний вбудований програмний код обробки помилок для відновлення після несправності та правильного виправлення внутрішнього і зовнішнього стану.

У роботі показана ефективність на кількох серверних програмах, включаючи веб сервер та бази даних.

Експериментальні результати, як з реальними помилками, так і з ін'єкціями штучних помилок показали, що запропонований механізм може бути використано для відновлення виконання у більшості розглянутих випадків з невиликими операційними витратами. Використання неоптимізованого прототипу, який повністю автоматизує процес відновлення програмного забезпечення, займає всього десятки секунд, а замовлення на відновлення швидше, ніж поточне розгортання усунення помилок людиною. Крім того, не вимагається наявність вихідного коду програми. Кінцевим результатом є автоматичне відновлення програмних служб з невідомих і непередбачених програмних збоїв.

Було окреслено межі віртуалізації помилок за допомогою точок відновлення, запропонувано нову техніку самовідновлення програмного забезпечення для відновлення після помилок використано цільові системи для виявлення помилок програмного забезпечення в програмах, викликаних втручанням з метою використання вразливостей програмного забезпечення, і отримання кінцевого стека викликів. Це узгоджується з потенційним набором точок відновлення шляхом відкату та повторення виконання з помилкою, щоб визначити, яку точку відновлення можна використовувати для відновлення після помилки.

Система динамічно виправляє запущену робочу програму до точки самоперевірки в точці відновлення та, якщо виникає помилка, повертається до

точки відновлення і повертає відоме значення, якщо використовується для відповіді на неправильний вхід, який використовується власними вбудованими програмами у механізмах обробки помилок для відновлення після несправності.

Результати роботи можуть набути поширення та застосовуватись у розробці більшості програмних, інформаційних систем з метою автоматизації відновлення, налагодження та експлуатації сучасних комп'ютерних та хмарних систем.

СПИСОК ЛІТЕРАТУРИ

1. Волк М.О., Лунічкін О.Г. Комп'ютерні системи з самовідновленням. Системи управління, навігації та зв'язку, 2022, випуск 1(67), с. 48-51
2. Sullivan, M., Chillarege, R.. Software Defects and Their Impact on System Availability–A Study of Field Failures In Operating Systems. In Proceedings of the 21st International Symposium on Fault-Tolerant Computing (FTCS-21), June 2021. pages 2–9,
3. Kolettis, N., Fulton, N. D. Software Rejuvenation: Analysis, Module and Applications. In Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS- 25), pages 381–395, June 2019.
4. Candea, G., Fox, A.. Crash-Only Software. In Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), pages 12–20, May 2013.
5. King, S. T., Dunlap, G. W., Chen, P. M.. Debugging Operating Systems With Time-Traveling Virtual Machines. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX 2015), pages 1–15, Apr. 2015.
6. Bressoud, T. C., Schneider F. B. Hypervisor-Based Fault Tolerance. In Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP 1995), pages 1–11, Dec. 1995.
7. Paxson, V.A System For Detecting Network Intruders In Real-Time. Computer Networks, 31(23-24):2435–2463, Dec. 2019.
8. Norton, M., Roelker, D. Snort 2.0 Protocol Flow Analyzer. Sourcefire White Paper, Apr. 2014.
9. Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., Stolfo, S. J. On the Infeasibility of Modeling Polymorphic Shellcode. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS 2017), pages 541–551, Oct. 2017.
10. Costa, M., Crowcroft, J., Castro, M., Rowstron, A., Zhou, L., Zhang, L., Barham., P. Vigilante: End-To-End Containment of InternetWorms. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2005), pages 133–147, Dec. 2015.
11. Qin, F., Tucek, J., Sundaresan, J., Zhou. Y., Rx: Treating Bugs As Allergies—A Safe Method To Survive Software Failures. In Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP 2015), pages 235–248, Oct. 2015.
12. Chandra, S. An Evaluation of the Recovery-Related Properties of Software Faults. PhD thesis, University of Michigan, Sept. 2000.
13. Song, Y., Locasto, M. E., Stavrou, A., Keromytis, A. D., Stolfo S. J. On the Infeasibility of Modeling Polymorphic Shellcode. In Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS 2017), pages 541–551, Oct. 2017.
14. Tucek, J., Newsome, J., Lu, S., Huang, C., Xanthos, S., Brumley, D., Zhou, Y., Song., D. Sweeper: A Lightweight End-To-End System For Defending Against Fast Worms. In Proceedings of the 2nd European Conference on Computer Systems (EuroSys 2017), pages 115–128, Mar. 2017.
15. Demsky, B., Rinard., M. Automatic Detection and Repair of Errors In Data Structures. In Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA 2003), pages 78–95, Oct. 2021.
16. Rinard, M., Cadar, C., Dumitran, D., Roy, D. M., Leu, T., William J., Beebee, S. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI 2014), pages 303–316, Dec. 2014.
17. Sidiroglou, S., Locasto, M. E., Boyd, S. W., Keromytis, A. D. Building A Reactive Immune System For Software Services. In Proceedings of the 2015 USENIX Annual Technical Conference (USENIX 2015), pages 149–161, Apr. 2015.
18. Волк М.О. Журналізація станів програмних розподілених моделей та її використання в оптимістичних алгоритмах синхронізації. Збірник наукових праць Харківського університету Повітряних Сил. 2010, випуск 1 (23). С.104–107.
19. Рубан І.В., Волк М.О., Рісухін М.В. Метод самовідновлення розподіленого програмного забезпечення в гетерогенних комп'ютерних системах. Телекомунікаційні та інформаційні технології. 2019. № 3 (64), с. 17-23

Received (Надійшла) 14.03.2023
Accepted for publication (Прийнята до друку) 02.05.2023

Self-healing computer systems

Maksym Volk, Maksym Hora, Vladyslav Labazov, Andriy Mishchenko, Anton Barsukov, Vladyslav Holetz

Abstract. The article deals with the self-healing system of parallel software using journaling of recovery points. Self-healing is a necessary feature of modern software, which provides the possibility of automatic detection, diagnosis and restoration of system performance. The main stages of recovery are storing the state of programs (journaling) in recovery points, monitoring the state of programs to detect errors, creating patches, restoring the state of programs to the corresponding recovery point. The structure of the system is proposed in the work, the algorithm of its functioning is described; issues of destination and virtualization of restore points are discussed; a description of the experimental software system and its application to recovery of common software systems is given. The results can become widespread and be used in the development of most software and information systems for the purpose of automating the recovery, debugging and operation of modern computer and cloud systems.

Keywords: computer system, self-healing, fault, backup, recovery point.