УДК 004.051

А.Ю. Кривцов

Національний аерокосмічний університет імені М.Є. Жуковського «ХАІ», Харків

ЗАСТОСУВАННЯ СТАТИЧНОГО АНАЛІЗУ ВИХІДНОГО КОДУ ДЛЯ ПІДВИЩЕННЯ ЕНЕРГОЕФЕКТИВНОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Розглянута задача підвищення енергоефективності програмного забезпечення. Основна увага приділена оптимізації програмного коду. Запропоновано використовувати статичний аналіз як основний інструмент у оптимізації. Проведено аналіз основних способів оптимізації програмного коду на базі експериментально отриманої інформації. Сформульовано висновки за результатами проведеного аналізу.

Ключові слова: *енергоефективність, енергоспоживання програмного забезпечення, статичний аналіз, оптимізація програм, програмна інженерія, якість програм.*

Вступ

Проблема енергоефективності є однією з найважливіших у сучасному світі. Зростаюча продуктивність апаратної частини інформаційних систем привела до потужного сплеску споживання енергії в робочих станціях і серверах. Особливо це позначилося на великих центрах обробки даних, в яких міститься багато обчислювальної техніки і мережевого устаткування. Крім того, проблема збільшення споживання енергії торкнулася сфери мобільних інформаційних технологій. Зростання апаратного енергоспоживання було швидшим, ніж розвиток технологій зберігання електроенергії, зокрема збільшення місткості акумуляторів. Цей факт виражається в істотному скороченні терміну служби акумуляторів мобільних пристроїв при розширенні їх функціональних можливостей.

Таким чином, проблема підвищення енергоефективності інформаційних технологій пов'язана не лише з необхідністю економії енергії, але і з необхідністю збільшення терміну служби акумулятора мобільних пристроїв.

Споживання енергії будь-якого пристрою залежить не лише від устаткування, але і від його програмного забезпечення [1, 2]. Таким чином, методи збільшення енергоефективності створюються як для апаратного, так і програмного забезпечення.

Методи, пов’язані з програмним забезпеченням в цілому оптимізують вищі рівні ієрархії проекту системи, таким чином, зрештою даючи істотніші результати, ніж апаратні методи.

У роботі [2] запропоновано цілісний підхід до створення зеленого програмного забезпечення, який проходить через увесь цикл його розробки. Він складається з п’яти етапів, включаючи в себе розробку вимог до програмного забезпечення, проектування, побудову, компіляцію та тестування. Етап побудови є найскладнішим і найбільш трудомістким. Він складається з чотирьох напрямів [2]:

– алгоритмічна оптимізація;
– оптимізація програмного коду;
– використання контекстної інформації;
– оптимізація ефективності простою.

Ми зосередимо свою увагу на оптимізації програмного коду, бо по-перше її методики є універсальними для більшості програмного забезпечення, а по-друге їх легко можна використовувати і для коду вже створених програм.

1. Оптимізація програмного коду

Серед практичних завдань програмної інженерії особливе місце займає завдання оптимізації роботи програмного забезпечення [3, 4]. При цьому під оптимізацією розуміється модифікація програмного забезпечення для поліпшення його ефективності. Слід зазначити, що термін «оптимізація програмного забезпечення» (чи «оптимізація програмного коду») не припускає строгого рішення задачі оптимізації (тобто знаходження екстремуму) в строгій математичній постановці [5, 6]. Оптимізоване програмне забезпечення зазвичай лише відповідає заданим обмеженням, при цьому, як правило, неможливо довести, що отримані характеристики відповідають екстремуму. Така ситуація пояснюється двома наступними причинами.

По-перше, функціонування програмного забезпечення залежить від вхідних даних, тобто оптимальне функціонування при одних вхідних даних не означає оптимального функціонування при усіх можливих вхідних даних.

По-друге, в більшості випадків неможливо задати вагу для компонентів програмного коду, оскільки усі вони мають бути виконані. У сучасній літературі по програмній інженерії відсутня яка-небудь математична формалізація постановки завдання оптимізації програмного забезпечення за критерієм енергоспоживання. Слід також відмітити, що публікації, присвячені теоретичним аспектам оптимізації ПЗ, досить рідкісні, і як правило, зводяться до прийомів роботи з

інструментальними засобами оптимізації. Завдання оптимізації програмного забезпечення може бути представлене як набір ітерацій, що включають перевірку того, чи відповідають після модифікації параметри енергоспоживання заданим і подальшу зміну структури коду для досягнення цього (рис. 1).

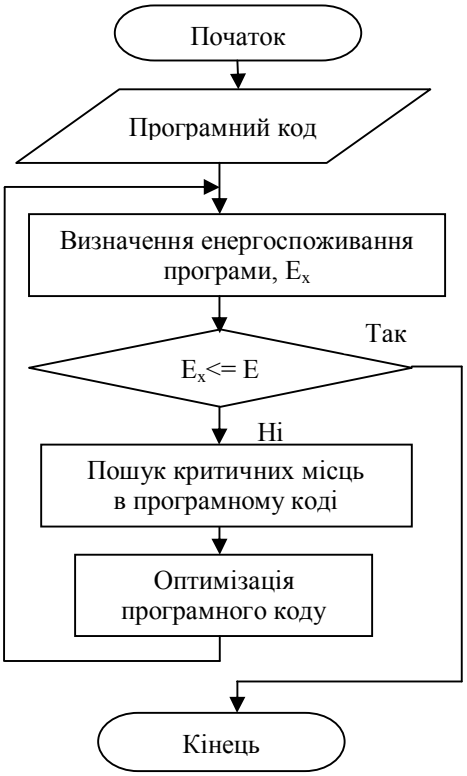

Рис. 1. Алгоритм оптимізації програмного коду

Формально, ми маємо завдання оптимізації, а в контексті цього дослідження – завдання мінімізації енергоспоживання. Таким чином, критерієм оптимізації є деяка функція:

$$E = \phi(x_1, x_2, ..., x_n) \rightarrow \min, \qquad (1)$$

де E – енергоспоживання; x – усі параметри (метрики), які прямим або непрямим чином можуть впливати на енергоспоживання системи.

Вибір усієї сукупності метрик, що впливають на енергоспоживання програми, є дуже індивідуальним для кожного типу програмного забезпечення. Як правило, сучасне програмне забезпечення це надзвичайно складна система з величезним числом зв'язків і залежностей. І справа не лише в складності самого програмного продукту. Будь-яка програма має середовище виконання мови програмування, функціонує в операційній системі, взаємодіє з іншими сервісами і програмним забезпеченням – усі ці елементи також є окремими складними системами з не меншим числом внутрішніх зв'язків і залежностей. Якщо ж додати ще і взаємні залежності метрик, то отримаємо неймовірне число всіляких комбінацій параметрів [7]. Проте є такі метрики, які незалежно будуть присутні у кожній програмі, наприклад, споживання енергії центральним процесором, оперативною пам'яттю та інше.

До оптимізації програмного коду пред'являються наступні вимоги [3]:

– оптимізація має бути максимально машинно-незалежною і переносимою на інші платформи (операційні системи) без додаткових витрат і істотних втрат ефективності. Ми повинні залишатися виключно у рамках цільової мови, причому, бажано використовувати тільки стандартні засоби і за всяку ціну уникати специфічних розширень, наявних тільки в одній конкретній версії компілятора;

– оптимізація не повинна збільшувати трудомісткість розробки (у тому числі і тестування) програмного забезпечення більш ніж на 10 – 15 %, а в ідеалі, усі критичні алгоритми бажано реалізувати у вигляді окремої бібліотеки, використання якої не збільшує трудомісткості розробки взагалі;

– оптимізований алгоритм повинен давати виграш не менше чим на 10 – 15 % в енергоспоживанні;

– оптимізація повинна допускати безнаслідкове внесення змін. Досить багато технік оптимізації деградують програму, оскільки навіть незначна модифікація оптимізованого коду може знищити всю оптимізацію.

Для проведення оптимізації потрібні спеціальні методики та інструменти. Одним із таких є статичний аналіз програмного коду.

Статичний аналіз коду – аналіз програмного забезпечення, що робиться (на відміну від динамічного аналізу) без реального виконання досліджуваних програм. В більшості випадків аналіз проводиться над якою-небудь версією програмного коду.

Залежно від використовуваного інструменту глибина аналізу може варіюватися від визначення поведінки окремих операторів до аналізу, що включає увесь наявний програмний код. Способи використання отриманої в ході аналізу інформації також різні - від виявлення місць, що можливо містять помилки, до формальних методів, що дозволяють математично довести які-небудь властивості програми (наприклад, відповідність поведінки специфікації).

Отримання метрик і статичний аналіз часто поєднуються, особливо при створенні вбудовуваних систем (software quality objectives) [8].

Останнім часом статичний аналіз все більше використовується у верифікації властивостей програмного забезпечення, використовуваного в комп'ютерних системах високої надійності, особливо критичних для життя. Також застосовується для пошуку коду, що потенційно містить уразливості [9].

Розглянемо переваги статичного аналізу коду. Статичні аналізатори дають повне покриття коду, тобто перевіряють навіть ті фрагменти коду, які отримують управління украй рідко. Такі ділянки

коду, як правило, не вдається протестувати іншими методами. Це дозволяє знаходити дефекти в обробниках рідкісних ситуацій, в обробниках помилок або в системі логування. Статичний аналіз не залежить від використовуваного компілятора і середовища, в якому виконуватиметься скомпільована програма. Це дозволяє знаходити приховані помилки, які можуть виявити себе тільки через декілька років [10].

Але треба враховувати і недоліки статичного аналізу коду. Він, як правило, слабкий в діагностиці витоків пам'яті і паралельних помилок. Щоб виявляти подібні помилки, фактично необхідно віртуально виконати частину програми. Програми статичного аналізу попереджають про підозрілі місця. Це означає, що насправді код може бути абсолютно коректний. Це називається помилково-позитивними спрацьовуваннями. Необхідність переглядати помилкові спрацьовування віднімає робочий час і послабляє увагу до тих ділянок коду, де насправді знаходяться помилки [10]. Крім того іноді існують проблеми з аналізом циклів, рекурсій та викликів глибокої вкладеності. Проте нові підходи (наприклад, розглянуті у роботі [11]) допомагають вирішувати ці проблеми.

Зазвичай для перевірки енергоспоживання програм використовують лише динамічний аналіз програмного забезпечення. Проте це не є раціональним.

Переваги статичного аналізу перед динамічним:

– статичний аналіз перевіряє увесь код. Він може перевірити навіть ті гілки, які украй рідко отримують управління. Виконати певні гілки програми в процесі її роботи буває дуже важко (складно підготувати дані, складно зімітувати аварійну ситуацію і так далі);

– динамічний аналіз часто пасує при тестуванні ресурсоємних програм. Якщо програма використовує гігабайти даних, то впровадження динамічного аналізатора може привести до неприпустимого зниження швидкості роботи. Статичний аналіз не залежить від етапу виконання, оскільки аналізує програмний код. При цьому швидкість його аналізу легко і просто масштабується;

– можна перевірити код у той момент, коли запуск певних тестів ще неможливий. Це означає, що ми усуваємо деякі помилки максимально рано, а значить максимально дешево;

– статичний аналізатор може виявити ситуації, абсолютно безпечні з точки зору динамічного аналізу.

2. Результати експерименту

Отже, перейдемо безпосередньо до експериментів. Експеримент повинен мати такі властивості:

– відтворюваність;

– репрезентативність;

– простота застосування;

– перевіряємість;

– повнота охоплення;

– точність.

Для проведення експериментів використовувалося наступне програмно-апаратне забезпечення:

– ноутбук Asus X55A (X55A – SX050D);

– процесор: IntelPentium B970 (2.3 ГГц);

– оперативна пам'ять: 4 Гб, DDR3 - 1066Гц;

– операційна система: Windows 7 Ultimate (x64);

– Intel Power Gadget 3.0.

В ході роботи було проведено ряд серій експериментів для визначення якості методів підвищення енергоефективності програмного забезпечення шляхом оптимізації програмного коду за допомогою статичного аналізу.

Під час експериментів за допомогою Intel Power Gadget 3.0 проводилися вимірювання споживання енергії центральним процесором та його поточної потужності. Споживання енергії центральним процесором як метрика присутнє у будь-якій програмі. При цьому треба врахувати, що енергоспоживання та потужність сучасних процесорів набагато більше ніж наприклад, у оперативної пам'яті.

У першій серії експериментів проводиться дослідження впливу передбачуваності переходів на енергоспоживання.

Переходи є однією з базових операцій будь якої мови програмування. Нажаль команди умовних переходів відносяться до найбільш проблемних з точки зору ефективності виконання процесором, оскільки можуть порушити черговість виконання команд. Процесор може коректно передбачити більшість переходів. Для того, щоб найкращим способом визначити, чи буде перехід коректно передбачатися, необхідно уявити собі еталон переходів. Якщо існує передбачуваний еталон, то процесор майже завжди зможе виявити його і правильно передбачити перехід. За відсутності передбачуваного еталону процесор в деяких випадках прогнозує невірний результат переходу і енергоефективність падає. Проте без прогнозування переходів процесору довелося б зупинятися на кожному переході і чекати багато тактів закінчення виконання переходу в конвеєрі процесора. В той же час споживання енергії при відключеному механізмі передбачування переходів і при відновленні після невірно передбаченого переходу приблизно однакові. Тому передбачування переходів - це завжди добре.

Важливо уникнути непередбачуваних переходів (особливо в критичному в плані енергоспоживання коді), щоб отримати максимальний виграш в енергоефективності.

Передбачуваність часто можна збільшити шляхом зміни черговості перевірки умов в коді і за рахунок простих змін коду.

З точки зору оптимізації переходи можна згрупувати в п'ять категорій:

– умовні переходи, що виконуються уперше;

– умовні переходи, які виконувалися більше одного разу;

– виклик і повернення;

– непрямі виклики і переходи;

– безумовні прямі переходи.

У експерименті було використано два варіанти коду інваріантних галужень:

```
//не оптимізований код
    int data = 0;
    Random S = newRandom();
    int value;
    for (int j = 0; j < 20; j++){
    value = S.Next(20);
    for (int i = 0; i < 1000000; i++){
    if (value> 10)
    data++;
    else
    data--;}}
//оптимізований код
    int data = 0;
    Random S = newRandom();
    int value;
    for (int j = 0; j < 20; j++){
    value = S.Next(20);
    if (value> 10){
    for (int i = 0; i < 1000000; i++)
    data++;}
    else{
    for (int i = 0; i < 1000000; i++)
    data--;}}
```

Результати проведених експериментів зображені на рис. 2 – 4. На цих рисунках і далі «Вар 1» – це не оптимізований варіант коду, «Вар 2» – оптимізований.

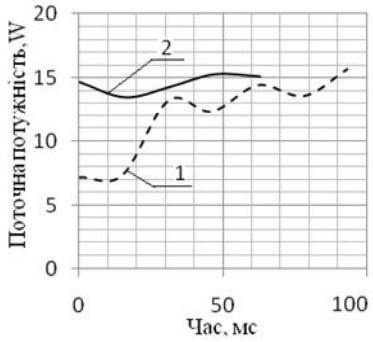

Рис. 2. Діаграма поточної потужності процесору під час роботи алгоритму:
1 – Вар 1; 2 – Вар 2

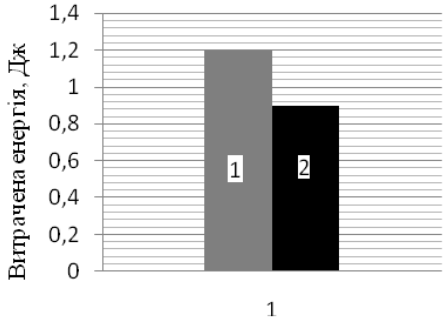

Рис. 3. Діаграма витраченої процесором енергії на виконання алгоритму:
1 – Вар 1; 2 – Вар 2

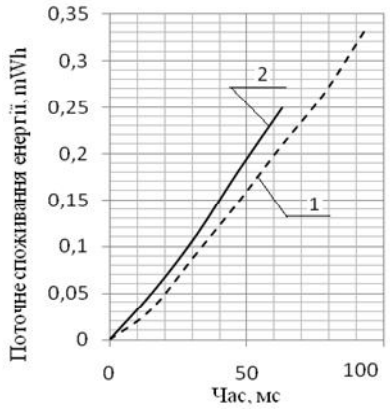

Рис. 4. Діаграма поточного споживання енергії процесором під час роботи алгоритму:
1 – Вар 1; 2 – Вар 2

Час відображений на діаграмах – це час виконання основного алгоритму програми, який наведено у роботі. Таким чином ми побачили, що хоч поточна потужність оптимізованого варіанту збільшилася, але за рахунок зменшення промахів переходів час роботи алгоритму зменшився. У результаті ми отримали виграш в енергії у 25 %. При цьому прискоривши роботу алгоритму на 32,26 %.

Друга серія експериментів присвячена роботі з оптимізацією доступу до пам’яті.

Витрати енергії на пам’ять пов’язані з операціями доступу до інструкцій, або до даних. Витрата енергії за операцію доступу залежить від об’єму пам’яті. Отже, для великих модулів пам’яті, що знаходяться поза процесором, енергоспоживання значно вище, ніж для пам’яті невеликого розміру, вбудованої в процесор. Ця складова споживання енергії так само залежить від самого виконуваного програмного забезпечення. Енергоспоживання пам’яті, використовуваної для зберігання інструкцій, залежить від розміру програмного коду, який визначає об'єм використовуваної пам’яті, і кількості виконуваних інструкцій, які визначають кількість операцій вибірки з пам’яті. Енергоспоживання пам'яті, використовуваної для зберігання даних, залежить від об’єму оброблюваних програмою даних, і від того, наскільки інтенсивно додаток працює з даними, тобто наскільки часто він дістає доступ до даних. Згідно типової моделі споживання енергії витрати на операції доступу до пам’яті, і витрати енергії прямо пропорційні кількості операцій доступу, розміру і кількості портів введення-виведення пам’яті, напруги живлення, і технології виробництва [11].

Згідно з вищесказаним, кількість виконуваних інструкцій має двократну дію на енергоспоживання – за допомогою виконання інструкцій за допомогою процесора, і витягання інструкцій з пам'яті. З іншого боку, вклад операцій доступу до пам’яті в енергоспоживання розглядається одноразово, як сам по собі. Але витрати електроенергії на роботу з даними в па-

м'яті набагато вище, ніж витрати на операції доступу до інструкцій, або енергія витрачена на виконання інструкцій процесором [11]. Рішення проблем пам'яті:

– вирішити усі виявлені проблеми випереджаючого запису;

– використовувати менше пам′яті, що скоротить промахи кеша, пов′язані з примусовим завантаженням;

– підвищувати ефективність кеша;

– читати пам′ять заздалегідь за рахунок передвибірки;

– писати в пам'ять швидше за допомогою команд прямого доступу до пам′яті.

У експерименті досліджувався програмний код множення матриці на матрицю. Для оптимізації був використаний метод доступу по адресах послідовно розташованих в пам'яті.

```
//не оптимізований варіант
        for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
        for (k = 0; k < N; k++){
        c[i, j] += a[i, k] * b[k, j];}}}
//оптимізований варіант
/*Доступ по адресах, послідовно розташованих
в пам'яті */
        for (i = 0; i < N; i++){
        for (k = 0; k < N; k++){
        for (j = 0; j < N; j++){
        c[i, j] += a[i, k] * b[k, j];}}}
```

Результати експериментів зображені на рис. 5 – 7.

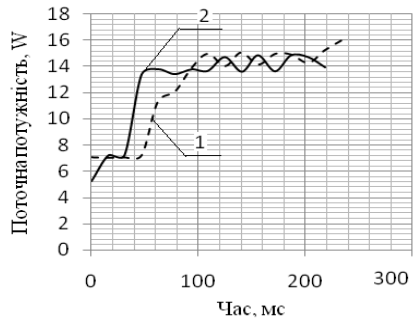

Рис. 5. Діаграма поточної потужності процесору під час роботи алгоритму:
1 – Вар 1; 2 – Вар 2

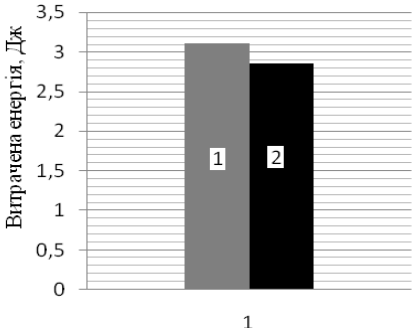

Рис. 6. Діаграма витраченої процесором енергії на виконання алгоритму:
1 – Вар 1; 2 – Вар 2

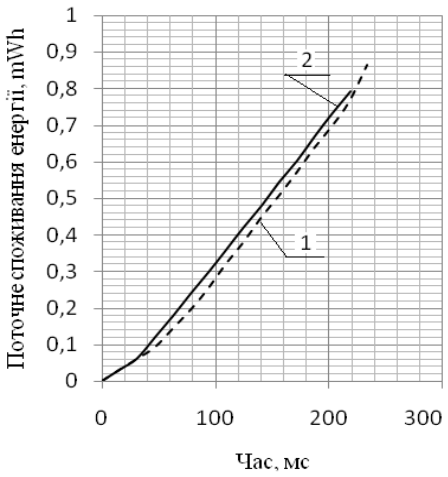

Рис. 7. Діаграма поточного споживання енергії процесором під час роботи алгоритму:
1 – Вар 1; 2 – Вар 2

В результаті ми побачили що завдяки простій перестановці двох строк отримали виграш в енергії на 8,1 %. При цьому прискоривши роботу алгоритму на 6,4 %.

Другий метод полягає у визначенні проблеми кеша в структурах даних.

```
//не оптимізований варіант
    #define  MAX_LAST_NAME_SIZE 16
    typedef struct TAGPH0NE_B00K_ENTRY{
    char LastName[MAX_LAST_NAME_SIZE];
    char FirstName[16];
    char email[16];
    char phone[10];
    char cell[10];
     char addrl[16];
    char addr2[16];
    char city[16];
    char state[2];
    char zip[5];
    _TAGPH0NE_B00K_ENTRY *pNext;}
    PhoneBook;
    PhoneBook *FindName(char Last[],
PhoneBook *pHead){
    while (pHead != NULL){
    if (stricmp (Last.pHead ->LastName) == 0)
return pHead ;
    pHead = pHead ->pNext;}
    return NULL;}
```

Кожна структура має розмір 127 байт, але тільки 20 байт задіються в кожному циклі пошуку. При такій організації марно втрачається 48 з 64 байт рядка кеша рівня 1 і 111 з 128 байт рядка кеша рівня 2, що для кеша рівня 1 дає ефективність максимум 25 %.

Для підвищення енергоефективності необхідно переробити структуру так, щоб усе змінні, в яких знаходяться прізвища, знаходилися в одному безперервному масиві, а інші менш затребувані дані – у іншому місці.

В результаті ці два масиви мають бути визначені таким чином:

```
//оптимізований варіант
    char LastNames[MAX ENTRIES *
MAX_LAST_NAME_SIZE];
    PhoneBook
PhoneBookHead[MAX_ENTRIES];
    PhoneBook * FindName(char Last[], char
*pNamesHead, PhoneBook *pDataHead, int
NumEntries){
    int i = 0; while (i <NumEntries){
    if (stricmp(Last.pNamesHead) == 0)
     return (pDataHead+i);
    i++;
    pNamesHead += strlen(pNamesHead)+1;}
    return NULL;}
```

Результати оптимізації представлені на рис. 8 – 10.

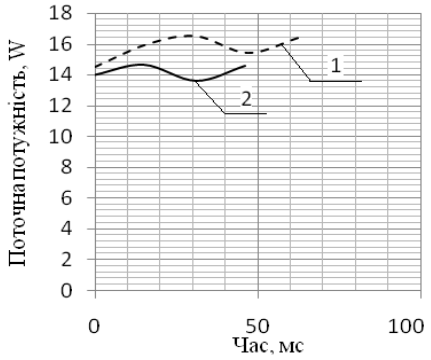

Рис. 8. Діаграма поточної потужності процесору під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

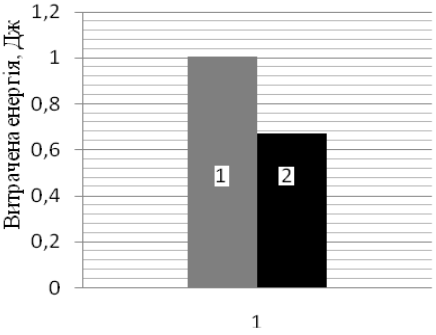

Рис. 9. Діаграма витраченої процесором енергії на виконання алгоритму: 1 – Вар 1; 2 – Вар 2

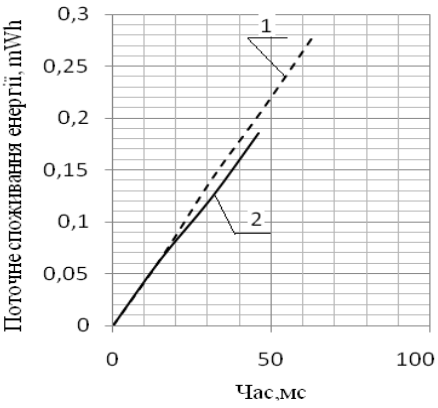

Рис. 10. Діаграма поточного споживання енергії процесором під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

В результаті експерименту ми бачимо що зменшилися як час виконання алгоритму, так і енергоспоживання. По енергоспоживанню ми отримали виграш в 33,53 %, а за часом в 32,35 %.

Серія наступних експериментів проводилася з оптимізацією циклів. Цикли є основним джерелом гарячих точок виключно завдяки ітераціям. Цикл сам по собі не обов'язково є вузьким місцем, а по деяких позиціях фактично навіть сприяє підвищенню продуктивності. ***По-перше,*** цикли зменшують кількість команд, що зберігаються. Програмам з меншою кількістю команд потрібно менше пам'яті, тому менше часу проводиться в очікуванні вибірки команд з основної пам'яті. ***По-друге,*** процесор керує декодовані команди, так що коли команда виконується повторно, виходить економія на часі декодування.

Чищення циклів це оптимізація циклу шляхом винесення з його тіла проблемних/надлишкових ітерацій. Простий приклад показаний в наступному фрагменті:

```
//не оптимізований варіант
    int p = 10;
    for (int i = 0; i < 10; ++i){
    y[i] = x[i] + x[p];
    p = i;}
//оптимізований варіант
    y[0] = x[0] + x[10];
    for (int i = 1; i < 10; ++i){
    y[i] = x[i] + x[i - 1];}
```

В ході експерименту ми отримали результати, наведені на рис. 11 – 13:

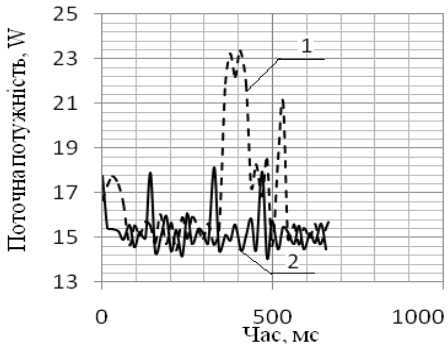

Рис. 11. Діаграма поточної потужності процесору під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

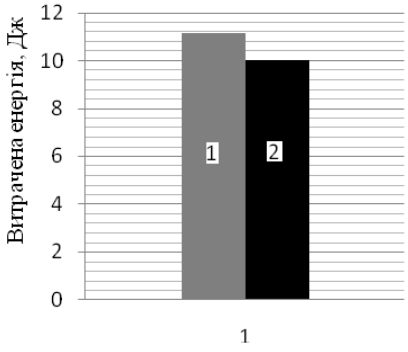

Рис. 12. Діаграма витраченої процесором енергії на виконання алгоритму: 1 – Вар 1; 2 – Вар 2

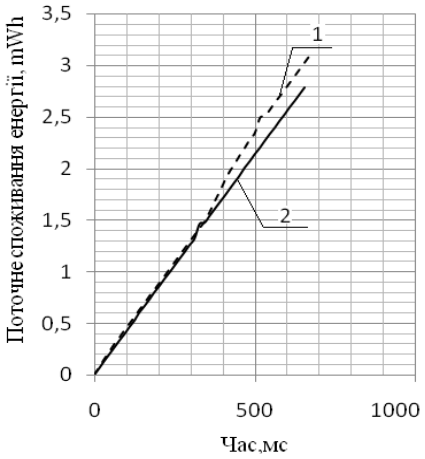

Рис. 13. Діаграма поточного споживання енергії процесором під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

В результаті експерименту ми бачимо що зменшилися як час виконання алгоритму, так і енергоспоживання. По енергоспоживанню ми отримали виграш у 10,25 %, а за часом на 2,38 %.

Наступним методом оптимізації циклів є розгортання циклів.

При розгортанні циклу відбувається об'єднання двох або більше ітерацій циклу і відповідне зменшення кількості проходів.

У розгорнутій версії циклу більше коду, зате виконується менше команд, що є джерелом витрат.

Продуктивність цього циклу залежить від стану кеша рівня 1, але взагалі розгорнута версія працює швидше, оскільки виконує менше команд, що є джерелом витрат. Проте в деякій точці виграш в продуктивності від скорочення кількості команд втрачається із-за додаткових витрат на вибірку і декодування більшої кількості команд.

Наприклад, не слід чекати великого виграшу від повного розгортання цього циклу в тисячу інструкцій.

```
//не оптимізована версія
        for (int i = 0; i < 1000; i=i+1){
        sum += y[i];}
//оптимізована версія
        for (int i = 0; i < 1000; i=i+4){
        sum += y[i];  sum += y[i + 1];
        sum += y[i + 2];  sum += y[i + 3];}
```

Результати експериментів представлені на рис. 14 –16.

В результаті експерименту ми бачимо що зменшилися як час виконання алгоритму, так і енергоспоживання. По енергоспоживанню ми отримали виграш на 10,3 %, а за часом на 9 %. Наступним методом є використання багатопоточності. Багатопоточність підвищить продуктивність і енергоефективність програмного забезпечення, оскільки робота одного потоку займає значно більше часу і енергії, чим будь-яка з багатопотокових комбінацій.

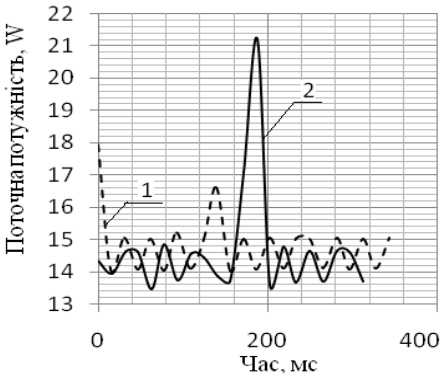

Рис. 14. Діаграма поточної потужності процесору під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

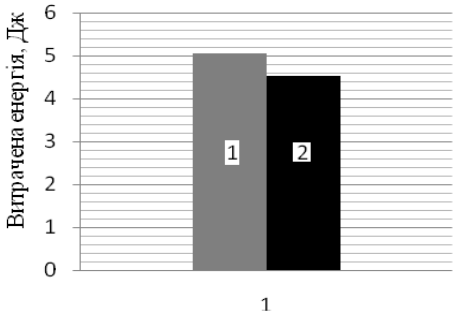

Рис. 15. Діаграма витраченої процесором енергії на виконання алгоритму: 1 – Вар 1; 2 – Вар 2

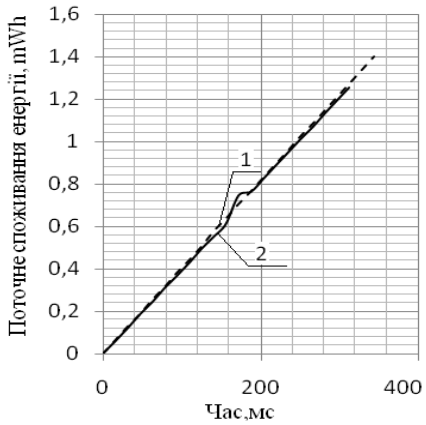

Рис. 16. Діаграма поточного споживання енергії процесором під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

При використанні прикладних програмних інтерфейсів (потокових в Win32 або Pthread в Linux) ця схема має один недолік, що ховається в низькорівневих деталях реалізації потоків. А саме: хтось повинен створювати потоки, відображувати їх на процесори, управляти ними, визначати, скільки треба створювати потоків, виходячи з ресурсів системи. Цей недолік несуттєвий, якщо питання собівартості і переносимості програмного забезпечення не важливі, оскільки є упевненість, що код виконуватиметься тільки на певній системі. На практиці ж, звичайно, для розробників важливі і собівартість, і переносимість, і зручність супроводу, і продуктивність. OpenMP вирішує усі ці

проблеми, привносячи до оптимізації паралельних програм і переносимість, і простоту.

Використовуючи потокові прикладні програмні інтерфейси в Win32 або Linux, можна отримати необхідну інформацію від операційної системи на етапі виконання і створити відповідну кількість потоків. Застосування OpenMP за допомогою компілятора звільняє від необхідності займатися низькорівневими деталями розбиття ітераційного простору, розподілу даних, планування потоків і синхронізації. З незначними зусиллями отримується та продуктивність, на яку здатні системи із загальною пам'яттю і декількома процесорами, з багатоядерними процесорами або з підтримкою гіперпоточності.

```
//не оптимізований код
        for (i = 0; i<SIZE; i++)
        for (r = 0; r<SIZE; r++){
        x = 0;
        for (j = 0; j<SIZE; j++)
        x += a[SIZE*i + j] * b[r + SIZE*j];
        c[SIZE* i + r] = x;}
        //оптимізований код
        #pragma omp parallel shared(a, b, c)
    private(x, i, j, r){
        #pragma omp for schedule (static)
        for (i = 0; i<SIZE; i++)
        for (r = 0; r<SIZE; r++){
        x = 0;
        for (j = 0; j<SIZE; j++)
        x += a[SIZE*i + j] * b[r + SIZE*j];
        c[SIZE* i + r] = x;}}
```

Результати експерименту представлені на рис. 17 – 19).

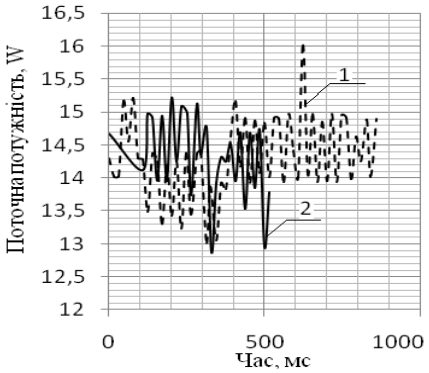

Рис. 17. Діаграма поточної потужності процесору під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

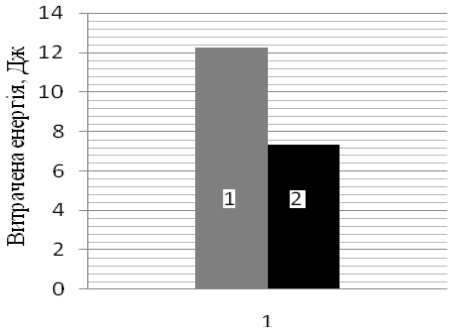

Рис. 18. Діаграма витраченої процесором енергії на виконання алгоритму: 1 – Вар 1; 2 – Вар 2

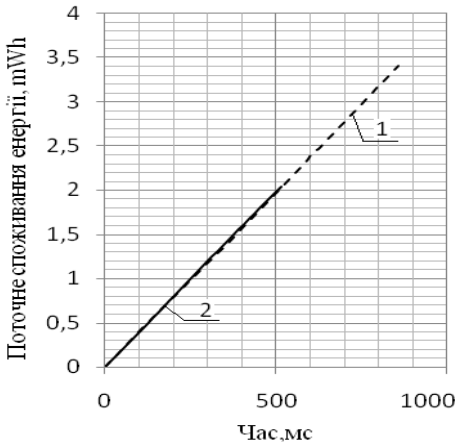

Рис. 19. Діаграма поточного споживання енергії процесором під час роботи алгоритму: 1 – Вар 1; 2 – Вар 2

Таким чином ми побачили, що хоч поточна потужність оптимізованого варіанту збільшилася(що обумовлене використанням усіх ядер), але за рахунок зменшення час роботи алгоритму загальне енергоспоживання зменшилося. У результаті ми отримали виграш в енергії на 40,12 %. При цьому прискоривши роботу алгоритму на 39,98 %.

Наведені вище методи дають змогу підвищити енергоефективність програмного забезпечення. Проте для того щоб провести оптимізацію програмного коду, потрібно спочатку знайти в ньому необхідні ділянки. Для цього нам потрібно використовувати статичний аналіз. Построївши поведінкову модель програми у вигляді графу потоку управління та використовуючи основні алгоритми статичного аналізу ми можемо швидко знаходити та оптимізувати необхідні ділянки програмного коду, таким чином автоматизувавши цей процес.

Висновок

Оптимізація енергоспоживання може виконуватися як на апаратному так і на програмному рівні. Для цього існує ряд підходів, що активно застосовуються. В той же час, все більший інтерес викликає оптимізація на рівні програмного забезпечення – написання енергоефективних застосувань.

У роботі були проаналізовані існуючі методи зниження енергоспоживання за рахунок оптимізації програмного забезпечення. В ході експериментів було практично доведено, що методи оптимізації програмного коду на основі статичного аналізу дають можливість підвищити енергоефективність програмного забезпечення.

Таким чином ми можемо використовувати статичний аналіз як інструмент для створення енергоефективного програмного забезпечення. Використовуючи цей інструмент, ми маємо можливість аналізувати та змінювати програмний код на будь-яких етапах створення програми, навіть на самих ранніх.

Крім того визначається необхідність формування та подальшого розвинення такого важливого напрямку досліджень, як створення зелених метрик безпосередньо для програмного коду. Тобто отримання більшої частини інформації про енергоспоживання програми без реального виконання, базуючись лише на його програмній структурі.

Список літератури

1. Fornaciari W., Gubian P., Sciuto D., and Silvano C. (1998), Power estimation of embedded systems: A hardware/software code sign approach, IEEE Trans. on VLSI Systems, Vol. 6/2, pp. 266 – 275.

2. Kryvtsov A.Y., and Hontovyi S.V. Approach to improve energy efficiency of information systems, // Радіоелектронні і комп'ютерні системи. – 2014. – № 1 (65). – С. 73 – 77.

3. Касперски К. Техника оптимизации программ. Эффективное использование памяти / К. Касперски. – БХВ-Петербург, – 2003. – 464 с

4. Магда Ю. С. Использование ассемблера для оптимизации программ на С++ / Ю. / С. Магда. – БХВ-Петербург, 2004. – 492 с.

5. Левитин А.В. Алгоримы: введение в разработку и анализ /А. В. Левитин. – М. : Вильямс, 2006. –576 с.

6. Кормен Т. Алгоритмы: построение ианализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест, К. Штайн.– М. : Вильямс, 2006. –1296 с.

7. Кривцов А. Ю. Аналіз зелених метрік та інструментів для оцінки енергофективності програмного забезпечення / А. Ю. Кривцов, С. В. Гонтовий // Радіоелектронні і комп'ютерні системи. – 2014. – № 5 (69). – С. 76 – 80.

8. Briand P., Brochet M., Cambois T., Coutenceau E., Guetta O., Mainberte D., Mondot F., Munier P., Noury L., Spozio P., and Retailleau F., (2010), Software Quality Objectives for Source Code., Proceedings Embedded Real Time Software and Systems Conference, ERTS2, Toulouse, France

9. Livshits B. (2006), Improving Software Security with Precise Static and Runtime Analysis, section 7.3 "Static Techniques for Security," - Stanford University , 229 p.

10. Макконнелл С. Совершенный код. Мастер-класс: пер. с англ. / С. Макконнелл. – М. : Издательско-торговый дом "Русская редакция"; СПб. Питер, 2005. – 896 с.

11. Юрченко А. В. Проектирование и анализ программного обеспечения с низким энергопотреблением с помощью программных метрик энергоэффективности / А. В. Юрченко // Наука и образование. – 2013. – № 1. – С. 215 – 234.

References

1. Fornaciari W., Gubian P., SciutoD., Silvano C. (1998), Power estimation of embedded systems: A hardware/software code sign approach, IEEE Trans. on VLSI Systems, Vol. 6/2, pp. 266 – 275.

2. Kryvtsov A.Y., Hontovyi S.V., (2014), Approach to Improve Energy Efficiency of Information Systems, Radioelektronni i komp'juterni systemy, No. 1 (65), pp. 73 – 77.

3. Kaspersky K. Tehnyka optymyzacyy programm. Effektyvnoe yspol'zovanye pamjaty [Technique of Optimization Programs. Efficient use of Memory], (2003), S-Peterburg, BHV-Peterburg., Russian Federation, 464 p (In Russian).

4. Magda Ju.S. Yspol'zovanye assemblera dlja optymyzacyy programm na C++ [Using Assembler to Optimize C++ Programs], (2004), S-Peterburg, BHV-Peterburg, 492 p (In Russian).

5. Levytyn A.V. Algorytmy: vvedenye v razrabotku y analyz [Algorithms: an introduction to the design and analysis], (2006), Moscow, Vyl'jams, Russian Federation, 576 p. (In Russian).

6. Kormen T., Lejzerson Ch., Ryvest R., and Shtajn K.. Algorytmy: postroenye y analyz [Algorithms: Construction and Analysis], (2006), Moscow, Vyl'jams, RF, – 1296 p. (In Russian).

7. Kryvtsov A.Y., and Hontovyj S.V. Analiz zelenyh metrik ta instrumentiv dlja ocinky energofektyvnosti programnogo zabezpechennja [Analysis of green metrics and tools for evaluating energy efficiency of software], (2014), Radioelektronni i komp'juterni systemy, No. 5 (69), pp. 76 – 80.(In Ukrainian).

8. Briand P., Brochet M., Cambois T., Coutenceau E., Guetta O., Mainberte D., Mondot F., Munier P., Noury L., Spozio P., and Retailleau F. (2010), Software Quality Objective sfor Source Code., Proceedings Embedded Real Time Software and Systems 2010 Conference, ERTS2, Toulouse, France (In English)

9. Livshits B. Improving Software Security with Precise Static and Runtime Analysis, section 7.3 "Static Techniques for Security," Stanford doctoral thesis, Stanford Univ., 229 p, 2006.

10. Makkonnell S. Sovershennyj kod. Master-klass [Perfect code. Master Class], Per. s angl., M.,Yzdatel'sko-torgovij dom "Russkaja redakcyja",SPb., Pyter, 2005., 896 p. (In Russian).

11. Yurchenko A.V. Proektirovanie i analiz programmnogo obespecheniya s nizkim energopotrebleniem s pomoschyu programm-nyih metrik energoeffektivnosti [Design and analysis software with low power consumption by using software metrics of energy efficiency], Nauka i obrazovanie., 2013, №1, P.215- 234 (In Russian).

Надійшла до редколегії 28.08.2015

Рецензент: д-р техн. наук, проф. В.С. Харченко, Національний аерокосмічний університет імені М.Є. Жуковського «ХАІ», Харків.

ПРИМЕНЕНИЕ СТАТИЧЕСКОГО АНАЛИЗА ИСХОДНОГО КОДА ДЛЯ ПОВЫШЕНИЯ ЭНЕРГОЭФФЕКТИВНОСТИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

А.Ю. Кривцов

Рассмотрена задача повышения энергоэффективности программного обеспечения. Основное внимание уделено оптимизации программного кода. Предложено использовать статический анализ как основной инструмент оптимизации. Проведен анализ основных способов оптимизации программного кода на базе полученной информации. Сформулированы выводы по результатам проведенного анализа.

Ключевые слова: *энергоэффективность, энергопотребление программного обеспечения, статический анализ, оптимизация программ, программная инженерия, качество программ.*

THE USE OF STATIC ANALYSIS FOR SOURCE CODE TO IMPROVE ENERGY EFFICIENCY OF SOFTWARE

A.Yu. Kryvtsov

We consider the problem of increase of energy efficiency for software in this paper. The main attention is paid to optimizing the source code. To use static analysis as the primary tool in optimizing is proposed. The analysis of the main ways to code optimization based on experimentally obtained data was conducted. The conclusions based on the results of the analysis are formulated.

Keywords: *energy efficiency, energy consumption of software, static analysis, program optimization, software engineering, quality programs.*