Лабораторна робота
"Визначення швидкодії обчислювальної системи"

1. Теоретичні відомості.

Найпростіше порівнювати однотипні за архітектурою комп'ютери. Так для IBM PC подібних комп'ютерів, які є найпоширенішими в Україні, ми інтуїтивно відчуваємо різницю у швидкодії для ПК з процесором, наприклад, 300 Мгц, 500 Мгц, 800 Мгц. Дуже часто для пересічного використача це і є основна характеристика швидкодії. І це емпіричне знання є досить точним лише за умови того, що переважно використовуються для дому та офісу близька за параметрами комплектація комп'ютерів, окрім саме процесорів. Фахівець же знає, що на швидкодію обчислювальної системи, навіть для однотипних платформ, впливає об'єм оперативної пам'яті, параметри відеопідсистеми, тип матиринської плати тощо. Разом з тим, для задач з активним обміном із локальними дисками, наприклад, швидкість самих дисків та шинного інтерфейсу, використання контролерів з паралельним доступом і т.і. значно більше впливає на нарощування ефективності (для цих задач), ніж потужність процесора. Тому і дивне в очах непрофесіонала бачити, що файл-сервер комплектується значно менш потужним (і дешевшим) процесором, ніж ігровий ПК.

Зауважимо, що типово використача цікавить не сам комп'ютер (його комплектація з врахуванням десятків параметрів), а його використання, яке передбачає деяку інфраструктуру: локальна мережа, розподіленість ресурсів у мережі з їх спільним використанням тощо. Тому ми будемо використовувати не термін *комп'ютер* для оцінки швидкодії/ефективності, а термін *обчислювальна система* (ОбСист).

Отже, навіть на рівні однотипних систем їх порівняння не може бути таким вже прямолінійним і має враховувати ту чи ту специфіку використання ОбСист. Хороша ОбСист для одних задач може бути зайвою за вартістю і можливостями для інших, які не потребують всього арсеналу "дорогих" ОбСист.

Задача порівняння ОбСист стає досить актуальною для сучасного ринку інформаційних технологій (IT), прикметною рисою якого є швидке моральне старіння всієї інфраструктури IT. Ринок IT починає тиснути на використача в плані нових витрат, які часто є реакцією на вдало проведені маркетингові акції, а не фактичною потребою таких витрат. Врешті, навіть невеликі витрати на кожний комп'ютер компанії в рамках сотень і тисяч робочих місць у останній може потребувати значні кошти. Сама ж IT-індустрія не задовольняється тим, що ряд систем, пристроїв можуть довго використовуватися, - все вироблене має розкуповуватися вже сьогодні, і вас виробники мусять переконувати робити постійну, власне часто зайву для вас, модернізацію всього. В цій ситуації для використача – від домашнього до корпоративного – важливо мати деякий якісний орієнтир реального виграшу чи його відсутності.

Природним чином виділились такі підходи щодо оцінки швидкодії/ефективності ОбСист:

1. Порівняння за формальною швидкістю процесора ОбСист;
2. Оцінка ОбСист за системою (як правило фіксованих за множиною) тестів.

Порівняння за швидкістю процесора обчислювальної системи

1. Враховуючи, що із зміною тактової частоти певного типу процесора його архітектура не міняється, можна порівнювати процесори за швидкістю виконання групи команд, але цей час виразити не в астрономічних одиницях, а в кількості тактів відповідного процесора. Врешті від кількості тактів ми завжди можемо перейти до астрономічного часу. Отже, більш зручно подавати характеристику швидкості у витрачених тактах.

2. Якби програма складалася лише з однієї команди, то на цьому порівняння можна було б і закінчити. Але команд десятки, а із своїми різновидами (за формою адресації та типом операндів, зокрема) вже сотні. Аналізуючи код програм (тобто реально використані команди) ми можемо оцінити деякі середні характеристики команди для вибраної групи примінень чи для деякої пересічної програми.

- Наприклад, ми можемо у множині програм підрахувати суму добутку кожної команди на кількість тактів команди і ділимо на кількість команд, отримавши ***середню команду*** у тактах.
- Або спочатку підраховуємо частоту кожної команди (типово це у %%, де всі частоти команд – 100%) і такти кожної команди множимо на частоту (вагу команди у "типовій" програмі), взявши суму таких добутків отримуємо *середньозважену команду* або оцінку за Гібсоном. Наприклад, хай для одного процесора така команда у 7.25 такти, а для другого 8.19 тактів. Відповідно маємо оцінку швидкості двох процесорів без прив'язки до їх частот, адже процесори різних виробників можуть мати взагалі неспівпадаючі ряди частот.

Потрібно зауважити щодо поняття середньої команди наступне. Практично всі сучасні універсальні процесори мають мікроконвейєри команд, коли при виконанні поточної команди йде аналіз і, при можливості, і виконання ряду наступних команд. Тоді на лінійних частинах програм за один такт може виконуватися команда , а то і декілька, хоча сама команда має далеко не один такт. Ряд процесорів до тогож мають не один, а декілька конвейєрів. Наприклад, процесор Intel Itanium (64-бітний) виконує до 20 інструкцій одночасно,

має 10-рівневий конвейєр. Відповідно, в оцінках *середньої команди* реально мати взагалі величини, менші за такт.

Оцінка обчислювальної системи за тестами

При розробці (вибору серед готових) тестів потрібно визначитися у таких питаннях.

1. Які можливості ОбСист ми будемо оцінювати: спеціальні чи загальні, і, відповідно тестами виступають деякі спеціально підібрані множини програм:

- спеціалізований набір задач (офісні, чисельного аналізу, графіка, потокове відео тощо);
- універсльний набір задач, в який включають всі типові класи задач із урахуванням пропорцій кожного на ринку програмного забезпечення; в свою чергу, побудова такої виборки є самостійною проблемою.

2. Спосіб реалізації тестів:

- ми можемо розробити свій набір програм; перевага такої реалізації в тому, що володіючи кодом тесту ми можемо його адекватним чином адаптувати до різних платформ, чим, зокрема, розширимо діапазон застосування тесту;
- взяти вже готові програми (системи), що широко використовуються на ринку, навіть якщо вони не кращі за всіма параметрами. Є сенс віддавати перевагу тим системам, які мають реалізацію на декількох платформах. Наприклад, для тесту офісних задач природньо брати за основу MS Office.

3. Нарешті, потрібно визначитися із складністю тестів. Наприклад, питання "що є типовою задачею для офісного текстового процесору?" не є простим. До тогож сам процес набору тексту вимірювати зайве, бо будь-яка ОбСист буде перекривати швидкість оператора. Це мають бути деякі дії, які виконуються автоматично, наприклад, *автоформатування* у MS Word, перерахунок таблиці у MS Excel тощо.

Окремими питанням є: методика вибору тестів, побудова генеральної виборки, опитування експертів, проведення статистичних вимірювань, порівняння із зразком (бенчмаркінг), які ми не будемо розглядати.

2. Приклади тестів швидкодії/ефективності обчислювальних систем

Тест Linpack. Розроблений в Арагонський національній лабораторії. Система програмування тестів – Фортран. Вирішує системи лінійних алгебраїчних рівнянь з розмірністю 100x100, 1000x1000 і т.д. (тобто 10^nx10^n, n=2, 3,… із масштабуванням у відповідності можливостей сучасних систем). Одиниця виміру – Linpack-Mflops.

Тест Whetstone. Розроблений у національній фізичній лабораторії Великобританії. Система програмування тестів – Фортран. Розв'язує фіксовану кількість задач (в одній із версій 949), зкомпільованих компілятором Whetstone. Дає оцінку переважно ефективності роботи з плаваючею арифметикою. Одиниця виміру – Whetstone/c.

Тест Khornestone. Вимірює ефективність при вирішенні наукових та економічних задач. Передбачені тести (в одній із версій 22), зокрема, оцінюють швидкості CPU та звернення до локальних дисків ОбСист. Одиниця виміру – Khornestone/c.

Тест Dhyrstone. Передбачає ряд операцій з інтенсивним навантаженням на процесор, серед яких: електронні таблиці, статистичні розрахунки, обробка текстів, доступ до баз даних. Типовий тест для оцінки самого лише процесора. Також часто є частиною інших тестів (наприклад, Power Meter) щодо оцінки "чистої" швидкодії процесора.

Тести Power Meter (компанія DiagSoft) та **WinStoned** (компанія Ziff-Davis) є концептуально близькими**.** Тест Power Meter розроблявся для вимірювань однозадачних платформ (MS DOS), відповідно тест WinStoned – для задач у мультапрограмному середовищі MS Windows. Тести імітують виконання ряду задач та мають такі вагові коефіцієнти підзадач тесту (в одній із версій):

1) робота із СУБД, зокрема, включає такі вимірювання: час доступу до ОП – 0.2; текстові операції – 0.1; MIPS (мільйон цілочисельних інструкцій процесора на секунду) – 0.2; файлові операції – 0.35; операції сортування – 0.15;
2) робота з електронними таблицями: час доступу до ОП – 0.15; текстові операції – 0.3; файлові операції – 0.05; перетворення адрес ссилок – 0.2; розрахунок формул – 0.3;
3) робота з текстовим процесором: час доступу до ОП – 0.1; текстові операції – 0.45; контексний пошук – 0.1; файлові операції – 0.05; Dhyrstone процесора – 0.3;
4) робота графічної програми: час доступу до ОП – 0.1; файлові операції – 0.35; Dhyrstone процесора – 0.15; Whetstone процесора – 0.3; відображення піксела (точки) – 0.4.

Тести WinBench (Windows User Benchmarking) **та WUBench** (Windows User Benchmarking) тестують підсистеми комп'ютера: процесор, ОП, відеоадаптер, локальні диски та Windows-операції. Шкала, в якій відображаються результати вимірювань, будується так (в одній із версій):

1) за одиницю (мінімальне значення) береться "класичний" IBM PC/XT, або який-будь "старуватий";
2) за значення 10 у кожному підтесті беруться результати якогось сучасного поширеного комп'ютера відомого виробника;

3) середнє арифметичне виконання підтестів для конкретної ОбСист беруться за оцінку швидкодії.

3. Постановка задачі

Лабораторна робота передбачає розробку спрощеної системи тестів, без побудови генеральної виборки, опитування експертів, проведення статистичних вимірювань.

Необхідно розробити програму, яка вимірює кількість виконуваних базових операцій (команд) за секунду конкретною ОбСист (комп'ютер + ОС + Система програмування). Вимірювання "чистої" команди процесора не потрібне (як і є у реальних програмних комплексах, що типово розробляються на мовах високого рівня, часто навіть на платформенно незалежних) і фактично не має сенсу. Вибір системи програмування за критерієм "*яка з них генерує швидший код*" зайва, - виберіть ту з них, яка для вас найбільш зручна.

До **базового набору операцій** достатньо включити операції ***додавання, віднімання, множення та ділення*** для **кожного з базових типів даних** (символьний, варіанти цілого, дійсний тощо, як це є в тій чи тій мові чи системі програмування). Інші операції, команди та типи – за бажанням виконавця. Враховуючи, що для всіх типів процесорів характерне об'єднання об'єктної команди додавання та віднімання у одну команду за рахунок відповідної зміни знаку одного з операндів (але для зручності кодування на рівні командного набору ці операції фігурують як окремі команди), не можна проводити вимірювання лише для однієї з цих двох команд. Якраз на цих двох операціях легко побачити стабільність тесту. Ви побачите, що за формальною, здавалось би, подібністю додавання та віднімання, добитися однакового результату непроста задача і потребує деяких "тонких" моментів при програмуванні.

Точність результатів – 2%. Достатньо на рівні вимірювань для лабораторної вважати, що для коротких операцій (додавання/віднімання для цілих слів) кількість операцій приблизно відповідає тактовій частоті процесора комп'ютера. Від цієї величини беремо 2%, і це значення буде ±похибка між одноіменними результатами для серії запусків програми. Наприклад, при тактовій частоті у 500 МГц, 2% дорівнюватиме 10 МГц, або це приблизно 10 млн. коротких оп/сек; отже похибка між одноіменними результатами ±10 млн. оп/сек.

Програма має демонструвати стабільність вимірювань для серії запусків. Потрібно враховувати, що при роботі на платформі MS Windows під час вимірювання за тестом може початися процес свопінгу системи, тому в такому випадку сусідні вимірювання у серії можуть відрізнятися на порядок. Такі запуски потрібно виключати з розгляду.

Результати мають бути представлені у табличній формі з відображенням для кожного тесту:
1) назви команди/операції,
2) типу/формату даних,
3) кількості операцій за секунду (зайві знаки у мантисі для заданої точності не відображати),
4) лінійної діаграми значення швидкості у відсотках відносно самої швидкої команди/операції, яка береться за 100%,
5) значення у відсотках (можна округляти до цілого).

Наприклад, для тестів на базі C/C++ (базових типів цієї мови) таблиця може бути такою:

+	int	7.825654e07	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	100%
-	int	7.710049e07	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	100%
*	int	3.469522e07	XXXXXXXXXXXXXXXXX	45%
/	int	6.168039e06	XXXXX	8%
+	long	3.706285e07	XXXXXXXXXXXXXXXXXXXXXX	49%
-	long	3.701185e07	XXXXXXXXXXXXXXXXXXXXXX	49%
*	long	1.667828e07	XXXXXXXXXX	
..	… … ...	… …… …	…… …… …… …… ..... …… …… ..... …… ……	… ….
/	double	60.94732e06	XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX	78%

Для текстового режиму (консольного примінення 25 рядків по 80 позицій) всі результати повинні поміститися на одній сторінці і для нормальної читабельності вирівняні у колонках. Виділення кольором, різними шрифтами тощо – зайве (хіба що це "автоматично" досягається у системах з візуальним програмуванням); у лабораторній важлива якість тестування, що і є метою роботи. Всі інші ефекти – за бажанням виконавця, але враховується найперше якість тесту, і розвинений інтерфейс не може замінити точність тесту!

Тест **не повинен довго виконуватися**; нормально - до 1 хвилини для всіх вимірювань. При правильному проектуванні тесту та програми має бути локалізованим необхідний діапазон кількості операцій для заданої точності. Збільшення ж числа ітерацій для похибки у 2% буде зайвою, а для мультипрограмних середовищ навіть призведе до її накопичення.

4. Рекомендації щодо виконання роботи

1. Враховуючи, що програмування задачі не потребує роботи з портами вводу/виводу, програмами обробки переривань, студентом самостійно може вибиратися будь-яка система програмування, включаючи інтерпретатори, аби в ній підтримувалися функції вимірювання проміжків часу.

2. Фактично робота полягає у вимірюванні дуже маленьких величин, які програмними засобами можуть бути виміряні лише в разі накопичування. Необхідно для кожної операції та типу будувати циклічний процес з вимірюванням часу на початку циклювання та вкінці. Відповідно різниця цих значень, поділена на кількість виконаних ітерацій дає приблизний час для однієї команди, точніше ітерації, бо в цей час включені також накладні витрати самого циклювання. Отже необхідно окремо отримати час роботи пустого циклу, який надалі відніматиметься від часу кожного непустого циклу.

3. Враховуючи те, що в програмах є лише один спосіб побудови ітераційного процесу – цикли, їх використання для накопичення малих інтервалів часу потребує розуміння наступних моментів:

- Цикли складаються із ряду регулярних обчислень: 1) перевірки можливості виконання ітерації тіла циклу та 2) корекції значення змінної цикла (лічильника ітерацій). Коли тіло цикла буде складатися лише із однієї, та й ще "короткої" команди (операції), виходитиме так, що накладні витрати на підтримку цикла займуть більше часу, можливо на порядок, ніж саме тіло циклу. Фактично "грубим" інструментом (цикл) потрібно виміряти малі величини. Тобто таке тіло на фоні підтримки циклу може бути на рівні похибки, і результат вимірювань часу виконання для пустого циклу та непустого, але з однієї "легкої" команди (операції), співпадати чи відрізнятися лише десь у 10-му знакові після коми.
- До цих нюансів реалізації, власне, самих операторів циклювання додається реалізація конвейєра та паралелізму виконання команд у сучасних процесорах, коли одночасно виконується декілька команд, що для циклу може означати одночасне виконання тіла циклу (із одного оператора) та дій з підтримки подальшого циклювання.

Для того. щоб позбавитися цього ефекту для "надлегких" тіл циклів (додавання/віднімання цілих слів, зокрема, операції зсувів, побітової логіки) найпростіше штучно наростити тіло циклу операціями, які вимірюються, відповідним чином врахувавши це при розрахунку часу.

4. При "нарощуванні" тіла циклу потрібно враховувати ряд моментів. Перш за все, вони пов'язані із оптимізацією об'єктного коду сучасними компіляторами, що фактично означає необов'язкову відповідність дій у вхідній програмі діям у об'єкному коді. Зайвий код, що не призводить до зміни даних, просто викидається. Дуже легко розпізнаються та виправляються, зокрема, такі конструкції (у прикладах використана мова C/C++):

- виду *a=15+205,* або у загальному вигляді *змінна=константнтний_вираз,* підраховуються ще на етапі компіляції, і замість цілого арифметичного виразу у коді буде лише команда присвоювення, а не додавання. Розробляючи тест для таких "команд додавання" насправді вимірюються команда присвоювання.
- *for(long i=0; i<FLOAT_ITER; i++) { a=b+c; a=b+c; a=b+c; … a=b+c; },* в яких серія операторів *a=b+c* буде винесена за цикл та зведена лише до одного разу вживання *a=b+c,* а цикл, ставши пустим, буде анульований (редукція циклу).
- *for(long i=0; i<FLOAT_ITER; i++) { a1=b1+c1; a2=b2+c2; a3=b3+c3; … a40=b40+c40; },* в яких тіло циклу буде винесене за цикл *{ a1=b1+c1; a2=b2+c2; a3=b3+c3; … a40=b40+c40; },* а, ставши пустим, цикл буде анульований.

  Кодувати серію однотипних операторів краще так:

  for(long i=0; i<FLOAT_ITER; i++) { a1=b1+c1; a2=b1+c2; b1=a1+c3; … a20=b1+c4; }.

або використовуючи у виразах змінну цикла. Але потрібно мати на увазі, що такі варіанти мають ефект арифметичної/геометричної (в залежності від операції) прогресії. І, як результат, можливі втрата значимості чи переповнення. Таким чином, вибираючи число ітерацій та форму тіла цикла, необхідно подбати про нейтралізацію ефекту прогресії та редукції циклів.

5. Використання індексів для тестуємої операції, наприклад: *a[i]=b[i]+c[i],* також небажана, бо адресація у масивах потребує додаткових розрахунків, що буде спотворювати результат вимірювань операції, або індексні вирази потрібно включати у вимірювання часу виконання пустого циклу

6. При розробці коду головної програми, де будуть виконуватися/викликатися окремі підтести, потрібне уважне використання конструкцій ітерування за допомогою оператора *switch*. Справа в тому, що всі ітерації продивляються послідовно, і вимірювання часу за токою схемою:

```
gettime(StartTime);
switch( Oper)
{ case '+': …….      break;
  case '-': …….      break;
  …….
  case '/': …….      break;
}
gettime(EndTime);
```

в разі різних умов виконання гілок у операторі, можна отримувати різний час. Потрібно кодувати так:

```
switch( Oper)
```

```
{ case '+': gettime(StartTime1); ……. gettime(EndTime1); break;
  case '-': gettime(StartTime2); ……. gettime(EndTime2); break;
  ……                ……                ……
  case '/': gettime(StartTime4); ……. gettime(EndTime4); break;
}
```

Тобто для кожної гілки необхідно вимірювати час окремо.

7. Щодо результатів вимірювань, то вони залежать від 1) стилю програмування та 2) використання/не використання опцій оптимізації коду компілятором. Справа в тому, що операції мають ряд варіантів задання операндів та виразів з них, а для кожного варіанту своя кількість тактів виконання Це є нормальна річ і вам немає потреби досягти однакового результату з програмою іншого виконавця, якщо там у операції інший варіант операндів.. Наприклад, для операції додавання маємо:

- *v=v1+ v2,* де всі змінні розміщені у пам'яті;
- *v=v1+ v2,* де змінні всі чи частина у регістровій формі;
- *v=v1+ const;*
- *v= const + const;*
- *v=v1+ v2 + … + vN;* буде більше відповідати тестуванню роботи зі стеком, ніж самої операції.

У лабораторній користуйтесь тим варіантом кодування, який природний для вас.

8. Для найшвидших команд/операцій потрібне і більше число ітерацій, а для "повільних" – менше. Необхідне число (діапазон) ітерацій можна підібрати:

- експеремтально вручну. Але що буде задовільним для однієї ОбСист для більш швидшої може не давати належну точність (особливо це буде помітно на серії запусків);
- програмно, почавши із деякого значення, просуватися у бік збільшення числа ітерацій із кроком (наприклад, +10% до попереднього числа ітерацій) до того моменту, коли сусідні вимірювання не будуть виходити за похибку;
- методом дихотомії (половинного ділення), вибравши за початковий інтервал достатньо широкий діапазон, але він швидко звузиться до оптимального.

9. Прив'язка до тіків чи секунд у часових інтервалах значення не має, - необхідна точність досягається відповідною кількістю ітерацій циклювання чи значення для інтервального таймера. Пам'ятайте, що "зайвими" ітераціями результат не покращується, але в мультипрограмних платформах, навпаки, погіршується.

10. При роботі з інтервалами часу використовуйте тип *double* для проміжних розрахунків. Зокрема функція *clock()* повертає результат типу *clock_t* (синонім типу *long)*, але робота з цим типом при переході від тіків до секунд не дає потрібну точність.