Uniwersytet Warszawski
Wydział Matematyki, Informatyki i Mechaniki

Szymon Acedański
Nr albumu: 219909

Wykorzystanie sprzętowych liczników zdarzeń do oceny wydajności algorytmów

Praca magisterska
na kierunku INFORMATYKA

Praca wykonana pod kierunkiem
dra hab. Krzysztofa Diksa
Instytut Informatyki

Wrzesień 2009

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

Data Podpis kierującego pracą

Oświadczenie autora (autorów) pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

Oświadczam ponadto, że niniejsza wersja pracy jest identyczna z załączoną wersją elektroniczną.

Data Podpis autora (autorów) pracy

Streszczenie

W pracy zaproponowano wykorzystanie modelowania procesora do usprawnienia procesu oceniania zadań na zawodach algorytmicznych, co pozwala uniezależnić ocenę od konfiguracji sprzętowej komputera użytego do oceny. Ta i inne przedstawione w pracy cechy modelu procesora wskazują na przydatność zaproponowanej techniki do wykorzystania na zawodach algorytmicznych.

Opisano także eksperymenty polegających na powtórzeniu oceny I etapu XVI Olimpiady Informatycznej przy użyciu kilku środowisk i metod oceny. Ich wyniki nie wykryły przeszkód do wprowadzenia przedstawionej metody na zawody Olimpiady Informatycznej.

Słowa kluczowe

zliczanie instrukcji, wydajność algorytmów, limit czasu procesora, Olimpiada Informatyczna, zawody programistyczne, CPU performance counters, algorithms evaluation, CPU time limit, Olympiad in Informatics, programming competitions

Dziedzina pracy (kody wg programu Socrates-Erasmus)

11.3 Informatyka

Klasyfikacja tematyczna

D. Software
D.2. Software Engineering
D.2.8. Metrics

Tytuł pracy w języku angielskim

Using hardware event counters for evaluating performance of algorithms

Spis treści

There is nothing wrong with change,
if it is in the right direction.

Winston Churchill

Wprowadzenie

Od kilku lat obserwujemy stały wzrost popularności różnorodnych konkursów programistycznych. Co roku powstają nowe serwisy oferujące możliwość zmierzenia się w zawodach programistycznych. Jeszcze 8 lat temu, gdy zaczynałem swoją przygodę z algorytmiką, licealiści mogli startować głównie w Olimpiadzie Informatycznej, raz do roku można było wziąć udział w „Wielkiej Przesmycce", a konkurs „Pogromcy Algorytmów" ogłaszał swoją pierwszą edycję. Dziś zawodnicy mają do dyspozycji portale organizujące konkursy co 2 tygodnie, olbrzymie bazy zadań treningowych, a w mało której szkole średniej nie ma zainteresowanych Olimpiadą Informatyczną.

Gdy poszedłem na studia, już czułem się zarażony Olimpiadą, która tak bardzo pomogła mi w rozwijaniu zainteresowań, dawała motywację do zgłębiania zagadnień algorytmicznych. Ponieważ zawsze interesowały mnie aspekty techniczne programowania, sposób działania różnorodnych systemów informatycznych, zagadnienia projektowania, działałem w zespole technicznym Olimpiady. Moim zadaniem była administracja i modernizacja systemu sprawdzającego.

Od czasu do czasu przechodziły mi przez głowę pomysły na usprawnienie jego działania, a przez to zwiększenie jakości oceny zadań, i w ogólności zwiększenie wartości Olimpiady. Jeden z tych pomysłów postanowiłem przedstawić w niniejszej pracy, w której przedstawiam propozycję zastosowania modelu procesora do pomiaru czasu działania programów. W pracy skupiam się przede wszystkim na porównawczej analizie dotychczasowej metody oraz nowej.

Przedstawiona metoda pomiaru wydajności nie bazuje na badaniu, ile czasu zajmuje wykonanie programu na konkretnej maszynie, lecz ile potrzeba wykonać instrukcji wyrażonych w bardziej uniwersalnym niskopoziomowym języku, niezależnym od konkretnej konfiguracji sprzętowej komputera używanego do oceny programu. Taka miara jest znana pod nazwą *wydajności względnej* programu.

W pracy przedstawiam analizę przydatności zaproponowanego rozwiązania, prezentując jego wady i zalety, zachęcając do dyskusji. Pokazuję również wyniki eksperymentu polegającego na użyciu proponowanej techniki do oceny wszystkich nadesłanych rozwiązań ostatniej edycji Olimpiady Informatycznej.

Na początku charakteryzuję konkursy oraz bazy zadań, z którymi mogą się spotkać współcześni zawodnicy. Następnie przybliżam sposób przygotowania zadań na takie konkursy. Po tym wstępie staram się przedstawić możliwe technologie modelowania procesorów, które mogą być przydatne w kontekście zawodów algorytmicznych, oraz przedstawiam proponowane rozwiązanie. Na końcu przestawiam wyniki eksperymentów.

Życzę miłej lektury.

Szymon Acedański

A competitive world offers two possibilities.
You can lose. Or, if you want to win, you can change.
Lester Thurow

Rozdział 1

O konkursach programistycznych

Na początku chciałbym poświęcić kilka stron na opisanie środowiska, w kontekście którego polecam rozważać niniejszą pracę.

1.1. Przegląd popularnych konkursów algorytmicznych

Pod pojęciem *konkurs programistyczny* może kryć się wiele różnych rzeczy — na przykład uczestnicy takiego konkursu mogą się ścigać w tym, kto napisze program najładniejszy, najszybszy, najwygodniejszy, najkrótszy, najbardziej niezrozumiały, dający najlepszy wynik itd. Czasem dorzuca się do tego zawody, które polegają na zaprojektowaniu jakiegoś komponentu.

Mimo takiej różnorodności rozumienia pojęcia konkursu, w tej pracy skupiam się na konkursach *algorytmicznych*. Zadania w takich konkursach polegają na napisaniu programu, który w sposób wsadowy (bez graficznego interfejsu użytkownika) przetwarza dane i oblicza wynik zgodnie ze specyfikacją w treści zadania. Poniżej wypisałem bardziej znane konkursy programistyczne wraz z krótkimi ich opisami i odnośnikami do stron internetowych.

- Olimpiada Informatyczna — rozgrywana od 1993 roku polska olimpiada przedmiotowa dla uczniów szkół średnich. Mogą w niej również startować gimnazjaliści. Zawody są trzyetapowe. Pierwszy etap jest rozgrywany przez internet i trwa miesiąc. Zawodnicy mają do rozwiązania około pięciu zadań. Liczba uczestników w roku szkolnym 2007/08 wyniosła 1 000 [10]. Kolejne etapy odbywają się w warunkach kontrolowanej samodzielności. Podczas każdej z dwóch pięciogodzinnych sesji (zazwyczaj poprzedzonych sesją próbą) zawodnicy mają do rozwiązania po dwa lub trzy zadania. Finaliści i laureaci mogą liczyć m.in. na ułatwiony dostęp na studia wyższe. Runda finałowa służy także wybraniu reprezentacji Polski na konkursy międzynarodowe.

  Na Olimpiadzie Informatycznej zadania są oceniane w skali od 0 do 100 punktów w zależności od tego, dla ilu zestawów danych testowych program obliczył odpowiedź poprawnie i w wymaganym czasie, oraz od jego szybkości.

  Dopuszczalne języki programowania: C, C++, Pascal, Java (tylko w latach 2007/08 oraz 2008/09)

  Witryna olimpiady: `http://www.oi.edu.pl`

- Międzynarodowa Olimpiada Informatyczna — jak sama nazwa wskazuje, są to światowe zawody programistyczne. Podobnie jak nasza krajowa olimpiada, jest przeznaczona dla uczniów szkół średnich i gimnazjów. Zawody są jednoetapowe i przebiegają podobnie

do drugiego bądź trzeciego etapu Olimpiady Informatycznej. W 2009 roku w zawodach uczestniczyło 80 narodowych drużyn z całego świata.

Dopuszczalne języki programowania: C, C++, Pascal

Witryna olimpiady: `http://ioinformatics.org/`

- Bałtycka Olimpiada Informatyczna — swoim zasięgiem obejmuje kraje nadbałtyckie, również dla licealistów i gimnazjalistów.

  Dopuszczalne języki programowania: C, C++, Pascal

  Witryna edycji 2009: `http://www.csc.kth.se/contest/boi/`

- Środkowoeuropejska Olimpiada Informatyczna

  Dopuszczalne języki programowania: C, C++, Pascal

  Witryna edycji 2009: `http://www.ceoi2009.ro/`

- ACM International Collegiate Programming Contest — najbardziej znane na świecie drużynowe zawody programistyczne dla studentów. Organizowane na skalę światową.

  Zawody są trzyetapowe. Pierwsza faza — lokalna, służy każdemu uniwersytetowi do wyłonienia reprezentantów na zawody regionalne. W Polsce selekcji służą zazwyczaj Akademickie Mistrzostwa Polski w Programowaniu Zespołowym. Corocznie w zawodach regionalnych startuje ponad 7 000 tysięcy zawodników reprezentujących ponad 1 800 uniwersytetów z 88 krajów świata (dane z 2008 roku [1]). Do finału kwalifikuje się około 100 drużyn z całego świata.

  Każda runda trwa 5 godzin. Do rozwiązania jest co najmniej 8 zadań. Każda drużyna ma do dyspozycji jeden komputer. Przysyłane rozwiązania są oceniane na bieżąco, a drużyny informowane o wynikach. Niezaliczone rozwiązania można wysyłać ponownie. Wygrywa drużyna, która rozwiąże największą liczbę zadań. Remisy rozstrzyga się obliczając czas rozwiązywania zadań. Za każde przysłanie niepoprawnego rozwiązania nalicza się dodatkowo karę czasową.

  Dopuszczalne języki programowania: C, C++, Java

  Witryna zawodów: `http://acmicpc.org`

  Witryna Akademickich Mistrzostw Polski w Programowaniu Zespołowym (2008 rok): `http://amppz.cs.put.poznan.pl/`

- TopCoder — najpopularniejszy serwis, który prowadzi rozgrywki głównie przez internet. Prowadzi ligi oraz zawody programistyczne w wielu kategoriach, w szczególności algorytmiczne — Algorithm Competitions. Około 2 razy w miesiącu odbywają się regularne jednorundowe zawody, w których może wystartować każdy. Na ich podstawie na bieżąco przydziela się zawodnikom punkty rankingowe i ogłasza bieżącą listę rankingową. Dodatkowo dwa razy w roku odbywają się zawody TopCoder Open oraz TopCoder Collegiate Challenge.

  Forma tych pojedynków na TopCoderze odbiega nieco od zasad wymienionych do tej pory konkursów. Zawodnicy mają do dyspozycji specjalny program — Arenę — która służy do interakcji z serwisem. W momencie rozpoczęcia rundy zawodnicy otrzymują trzy zadania o różnej punktacji. Ilość punktów otrzymanych za poprawne rozwiązanie zależy od czasu jaki upłynął od momentu otwarcia treści zadania do przysłania poprawnego rozwiązania. Zawodnicy mają godzinę na przysyłanie rozwiązań. Po tej

części zawodów ma miejsce runda, w której zawodnicy mogą nawzajem oglądać sobie przysłane rozwiązanie i... szukać błędów w cudzym kodzie. Za znalezienie przykładu danych wejściowych, na których czyjś program nie działa, bądź daje niepoprawną odpowiedź, są dodatkowe punkty.

Dopuszczalne języki programowania: C++, Java, C#, Visual Basic

Witryna serwisu: `http://topcoder.com/tc`

- Potyczki Algorytmiczne — otwarty konkurs algorytmiczny popularny w wielu środowiskach w Polsce. Może w nim wystartować każdy. Organizatorami są: Uniwersytet Warszawski oraz firma ADB Polska.

  Wyniki sześciu rund internetowych, z których każda trwa jeden dzień (za wyjątkiem weekendowej), decydują o kwalifikacji 20 najlepszych do finału. Tenże jest rozgrywany na zasadach zawodów ACM, z tym, że zawody są indywidualne.

  Dopuszczalne języki programowania: C, C++, Pascal, Java

  Witryna konkursu: `http://www.konkurs.adb.pl`

- Poznań Open (kiedyś Otwarte Mistrzostwa Wielkopolski w Programowaniu Zespołowym) — otwarty konkurs rozgrywany na zasadach ACM, organizowany przez Uniwersytet im. A. Mickiewicza w Poznaniu. Jedna runda. Popularny w Polsce szczególnie wśród studentów i licealistów.

  Dopuszczalne języki programowania: C, C++, Pascal

  Witryna konkursu: `http://www.mwpz.poznan.pl`

Warto też w tym miejscu zwrócić uwagę na istnienie różnorodnych serwisów internetowych, które ułatwiają przygotowania do zawodów algorytmicznych. Gromadzą one zadania i umożliwiają użytkownikom przysyłanie swoich programów do oceny. Często problemy pochodzą z zeszłych prawdziwych konkursów algorytmicznych. Takie serwisy są niezastąpionym źródłem zadań do treningów dla wszystkich przygotowujących się do konkursów algorytmicznych. Poniżej wypisałem kilka bardziej znanych baz zadań:

- Sphere Online Judge jest serwisem stworzonym przez studentów Politechniki Gdańskiej, teraz obsługiwany przez firmę Sphere Research Labs. Zawiera ponad 1300 publicznie dostępnych zadań algorytmicznych. Umożliwia automatyczne sprawdzanie przysłanych rozwiązań. Jest popularnym portalem treningowym wykorzystywanym szczególnie przez olimpijczyków i studentów.

  Serwis obsługuje 40 języków programowania.

  Witryna serwisu: `http://www.spoj.pl`

- Univeristy of Valladolid Online Judge działa na podobnych zasadach, jak Sphere Online Judge. W swojej bazie posiada ponad 2500 zadań w języku angielskim i również każdy może przysyłać do oceny swoje rozwiązania.

  Serwis posiada w swojej bazie wszystkie zadania ze światowych finałów ACM ICPC od roku 2000.

  Witryna serwisu: `http://uva.onlinejudge.org`

- Młodzieżowa Akademia Informatyczna jest portalem związanym bliżej z Olimpiadą Informatyczną i dlatego posiada w swoich zasobach zadania z zeszłych Olimpiad Informatycznych, Olimpiad Informatycznych Gimnazjalistów i Potyczek Algorytmicznych. Również umożliwia użytkownikom automatyczne ocenianie ich rozwiązań.

  Witryna serwisu: `http://www.main.edu.pl`

1.2. Przygotowywanie zadań na Olimpiadę Informatyczną

Aby przedstawić trochę szerzej kontekst, w którym warto zastanawiać się nad sposobem pomiaru czasu działania programów, czyli w szczególności sposobem oceniania, chciałbym zawrzeć poniżej opis procesu przygotowania zadania na zawody Olimpiady Informatycznej.

Przez lata prowadzenia i przygotowywania Olimpiady, zespół jurorów opracował jednolity proces opracowania wszystkich zadań. Tak więc przygotowanie zadania składa się z następujących faz:

1. Redakcja — czyli przygotowanie treści zadania. Zadania dostarczane przez pomysłodawców są często w formie krótkiej notatki. Redaktor musi wymyślić i redagować historyjkę, opisać format danych oraz przygotować jeden mały test do umieszczenia w treści zadania. W efekcie powstaje gotowy dokument z treścią zadania. Przykładowy taki dokument znajduje się w dodatku A.

2. Opracowanie — to część, gdzie powstają testy do oceniania, testy dla zawodników, programy wzorcowe we wszystkich wymaganych językach programowania, programy niepoprawne bądź nieoptymalne. Opracowujący także ustala ograniczenia na rozmiar danych wejściowych, przygotowuje programy sprawdzające poprawność testów oraz poprawność odpowiedzi. Na koniec pisze tzw. *dokument opracowania*, w którym opisuje zaimplementowane rozwiązania oraz testy, podaje informacje o sposobie punktacji poszczególnych testów i proponowanych limitach czasowych. Przykładowy dokument opracowania znajduje się w dodatku A.

3. Weryfikacja — mająca na celu skontrolowanie poprawności wykonania opracowania. Osoba weryfikująca oprócz sprawdzenia, czy wszystko zostało wykonane, ma za zadanie napisać jeszcze jeden program implementujący rozwiązanie wzorcowe. Faza weryfikacji czasem jest wykonywana kilkukrotnie, szczególnie w przypadku wykonania znaczących zmian w treści lub opracowaniu, lub przed zmianą przeznaczenia zadania.

Od czasu do czasu wykonuje się inne rodzaje prac — na przykład jedną z faz opracowywania zadania na Międzynarodową Olimpiadę Informatyczną jest recenzja. Zadania wykorzystane na Olimpiadzie Informatycznej po jej zakończeniu są opisywane, a powstałe artykuły publikowane w „Niebieskich Książeczkach” [10].

Wykonanie każdej fazy jest osobnym zadaniem i najczęściej każda faza jest wykonywana przez inną osobę.

1.2.1. Sposób ustalania limitów czasowych

Tym, na co największy wpływ w przebiegu procesu opracowania ma proponowana przeze mnie metoda, jest dobór limitów czasowych. Do tej pory limity te nie są ostatecznie ustawiane przez opracowujących ze względu na brak możliwości uruchomienia przez nich programów wzorcowych w środowisku oceny. Dobór limitów na podstawie szybkości zmierzonej na

własnych komputerach opracowujących nie ma sensu ze względu na zbyt duże różnice względem systemu oceniającego. A zatem opracowujący w dokumencie przygotowuje wskazówki dotyczące sposobu ustawienia limitów, a operator systemu sprawdzającego postępuje zgodnie z nimi w momencie umieszczania zadania w serwisie.

Większą uwagę warto jednak zwrócić na cele, jakim ma służyć wyznaczenie dobrego zbioru testów wraz z limitami czasowymi, i zadać pytanie, czy nowa metoda pomoże czy przeszkodzi w ich osiągnięciu. Podręcznik jurora Olimpiady Informatycznej [14] zawiera następujące wytyczne w tej kwestii:

> Testy powinny być tak dobrane, by różnicowały poszczególne klasy (poprawnych) rozwiązań. Dobierając testy należy kierować się następującymi wytycznymi:
>
> - To ile punktów zdobywa dane rozwiązanie powinno odzwierciedlać wysiłek (koncepcyjny i programistyczny) konieczny do jego opracowania. W przypadku wątpliwości należy równomiernie rozdzielić między coraz efektywniejsze rozwiązania.
> - To ile testów przejdzie dane rozwiązanie powinno zależeć jedynie od rodzaju rozwiązania, a nie od języka programowania, w którym jest zaimplementowane. (Stąd wymóg implementacji rozwiązań wzorcowych i mniej efektywnych w różnych językach programowania.)
> - Wszystkie rozwiązania poprawne, również te mniej efektywne, powinny zdobywać przynajmniej 30%–60% punktów. Dokładna wartość zależy od tego, na ile istotna w rozwiązaniu zadania jest efektywność, a na ile poprawność.
> - Rozwiązania o różnej złożoności powinny być rozróżniane minimum przez dwa testy (nie zgrupowane razem).
> - Rozwiązania niepoprawne (np. heurystyki, lub opierające się na błędnych założeniach) powinny otrzymywać mało punktów (co najwyżej 10%). W szczególności rozwiązania, które udzielają zawsze tej samej odpowiedzi (np. `0`, `NIE`, czy odpowiedź dla testu przykładowego) powinny otrzymywać 0 punktów (można w tym celu zastosować grupowanie testów).
> - To, które testy przechodzą rozwiązania niepoprawne nie powinno zależeć od ich złożoności czasowej (bo na tę wpływa język programowania), ale sama poprawność wyników i/lub zapętlenie się algorytmu.

Choć nie jest to wprost napisane w powyższych wytycznych, naturalną zasadą jest, że (poza ekstremalnymi przypadkami) dwa rozwiązania o tej samej złożoności powinny dostać taką samą liczbę punktów. Oczywiście nie zawsze można to zapewnić.

Po takim wstępie pozwolę sobie przejść do opisania proponowanej metody, a następnie przedstawię wyniki przeprowadzonych eksperymentów.

Innovation distinguishes between a leader
and a follower.

Steve Jobs

Rozdział 2

Po co i jak modelować procesor?

2.1. Problemy do rozwiązania

Tradycyjna metoda mierzenia czasu działania programów posiada szereg problemów, których sporą część wymieniam poniżej. Jak się później okaże, większość z nich może rozwiązać zastosowanie modelu procesora zamiast procesora fizycznego przy ocenie zadań na zawodach.

Nieprzenośność limitów czasowych

Nieraz po obejrzeniu wyników zawodów uczestnicy pytają, dlaczego ich program w systemie sprawdzającym działał dużo wolniej niż na ich komputerach domowych, i to na dodatek tylko w jednym zadaniu. Okazuje się, że czas działania programu zależy w praktyce od wielu czynników takich jak konkretna wersja użytego kompilatora czy model procesora. W pracy można zobaczyć dwa „spektakularne" przykłady takich różnic:

- W punkcie 3.2 przedstawiono obrazowy przykład programu, którego czas działania może się różnić ponadsześćdziesięciokrotnie na dwóch różnych procesorach o podobnej częstotliwości zegara.
- Później w punkcie 3.3.8 przedstawione jest zadanie, w którym rozwiązanie wzorcowe działa na procesorze taktowanym zegarem 3GHz siedmiokrotnie wolniej niż na innym procesorze dwugigahercowym.

Co więcej, wprowadzenie metody pomiaru czasu działania programów, która jest niezależna od sprzętu, pozwoliłaby już na etapie opracowania zadania precyzyjnie określić limity czasowe dla poszczególnych testów. Dotychczas ma to miejsce w momencie wgrywania opracowanego zadania do systemu sprawdzającego, czyli zazwyczaj niedługo przed zawodami. Czasem okazuje się, że spełnienie wymagań opracowującego dotyczących limitów czasowych nie jest możliwe. Wtedy w pośpiechu dokonuje się modyfikacji w opracowanym zadaniu, co skutkuje większą liczbą pomyłek i problemów podczas zawodów.

W przypadku wymiany maszyn sprawdzających, konieczne jest ponowne ustawienie limitów czasowych dla wszystkich zadań, których rozwiązania będą oceniane. Dla zawodów typu Olimpiada Informatyczna nie jest to dużą przeszkodą, gdyż raz zakończone zawody nie są powtórnie oceniane. Jednakże dla serwisów treningowych, będących zbiorami zadań, do których rozwiązania można wysyłać w dowolnym momencie, wymiana komputerów sprawdzających wiąże się z olbrzymim nakładem pracy na ponowne ustawienie limitów czasowych.

Ponadto opracowanie takiego sposobu oceny rozwiązań, który jest niezależny od konfiguracji sprzętowej maszyn oceniających, pozwoliłoby wielokrotnie wykorzystywać raz opracowane

zadania — na różnych zawodach, czy też w serwisach treningowych — bez konieczności wielokrotnego ustawiania limitów czasowych oraz zapewniałoby, że we wszystkich tych miejscach sposób oceny jest jednakowy.

Jednolita konfiguracja maszyn sprawdzających

Maszyny, na których odbywa się sprawdzanie, muszą mieć bardzo zbliżoną konfigurację sprzętową — w szczególności dokładnie takie same procesory oraz pamięć RAM o tej samej szybkości dostępu. W przypadku awarii jednej z maszyn sprawdzających, wymiana jej na identyczną wiąże się z dużymi kosztami, a nawet może być niemożliwa ze względu na brak komponentów o ściśle wymaganych parametrach.

Podobne koszty wiążą się z powiększeniem floty maszyn sprawdzających. Często ta operacja wymaga po prostu wymiany wszystkich maszyn. Takie rozwiązanie trudno nazwać skalowalnym.

Ponadto na maszynach sprawdzających zazwyczaj uruchomiona jest minimalna możliwa liczba usług, aby nie zakłócać przebiegu sprawdzania programu. Z tego względu maszyny pozostają nieużywane wtedy, gdy nie odbywają się żadne zawody.

Roczny koszt utrzymania maszyn sprawdzających OI przy pobieżnych szacunkach wynosi ponad 1 000 zł (tabela 2.1). Cena kilowatogodziny obejmuje koszty przesyłu i została obliczona na podstawie taryfy RWE S.A. (dostawcy prądu w Warszawie) dla przedsiębiorstw obowiązującej we wrześniu 2009 r.

Pobierana moc jednego komputera	100 W
Ilość komputerów	4
Cena prądu	37 gr/kWh
Roczny koszt prądu (bez VAT)	**1296 zł**

Tablica 2.1 Szacunkowy koszt utrzymania komputerów sprawdzających OI przez rok.

Niekompatybilność z wirtualizacją

W dzisiejszych czasach gwałtownie rozwijają się technologie wirtualizacji, pozwalające uruchomić na jednym fizycznym komputerze wiele wirtualnych maszyn, z których każda działa podobnie do zwykłego komputera. Wykorzystanie tej technologii pozwala znacznie obniżyć koszty utrzymania infrastruktury serwerów, których średni poziom wykorzystania często jest bardzo niski, przez konsolidację wielu fizycznych maszyn na jednym serwerze maszyn wirtualnych.

W przypadku Olimpiady Informatycznej, wirtualizacja maszyn sprawdzających pozwala również rozwiązać problem współdzielenia komputerów sprawdzających między wiele konkursów organizowanych przy użyciu sprzętu OI. W szczególności jedna maszyna sprawdzająca nie może być jednocześnie używana przez system OI oraz Olimpiadę Informatyczną Gimnazjalistów. Konieczna jest ręczna zmiana konfiguracji w przypadku zmiany jej przeznaczenia.

Niestety maszyny wirtualne nie są od siebie całkowicie niezależne, w szczególności działają wciąż na jednym fizycznym komputerze, a pamięć podręczna procesorów jest współdzielona między wszystkie maszyny wirtualne uruchomione na tym komputerze. Specyfiką wielu programów ocenianych na zawodach algorytmicznych jest to, że szybkość ich działania silnie zależy od rozmiaru dostępnej pamięci podręcznej. W takim przypadku pomiary czasu działania programów wykonane na maszynie wirtualnej są obarczone dużymi wahaniami, które

zależą od tego, w jakim stopniu inne maszyny wirtualne wykorzystują pamięć podręczną procesora.

Za jaką cenę?

Wszystkie powyższe problemy w mniejszym lub większym stopniu może rozwiązać wykorzystanie modelowania procesora do pomiaru „czasu działania" programu. Oczywiście wiąże się to także z pewnymi problemami, które staram się podsumować poniżej.

1. Każdy model procesora różni się jednak swoim działaniem od rzeczywistego procesora. Im wierniejszy model chcemy stworzyć, tym wolniej będzie działać taki „sztuczny" procesor. Ten trend będzie prawdziwy niezależnie od wybranej techniki modelowania.

   W rozdziale poświęconym eksperymentom starałem się przedstawić w sposób ilościowy różnice pomiędzy prostym modelem procesora, a różnymi rzeczywistymi procesorami. Przeprowadziłem również analizę porównawczą takich samych programów, lecz skompilowanych różnymi wersjami kompilatora.

2. Zmiana tak fundamentalnego elementu Olimpiady Informatycznej, jak sposób oceny zadań, na pewno będzie się wiązał z protestami wielu osób w środowisku, a co najmniej wywołałby burzliwą dyskusję. Wykonanie tego kroku wymaga skrupulatnych przygotowań i determinacji ze strony osób zaangażowanych w organizację Olimpiady Informatycznej.

Na końcu pracy powrócę raz jeszcze do powyższych problemów podsumowując je w kontekście wykonanych eksperymentów. Teraz natomiast chciałbym opisać sposoby modelowania działania procesora.

2.2. Modelowanie działania procesora

Jak już wspomniałem na początku poprzedniego rozdziału, rozwiązaniem wymienionych problemów może być zastąpienie rzeczywistego procesora jego modelem. To rozwiązanie jest również nazywane *maszyną wirtualną*. To określenie w języku polskim jest niejednoznaczne. W szczególności oznacza program, który potrafi programowo realizować zadania procesora, czyli potrafią interpretować pewien język maszynowy. Często ów język jest stworzony specjalnie do uruchamiania na maszynie wirtualnej, jest bardziej rozbudowany niż języki wykonywane przez sprzętowe procesory, albo jest stworzony do specyficznych zastosowań. Przykładami maszyn wirtualnych, w opisywanym znaczeniu, jest wirtualna maszyna Javy oraz wirtualna maszyna .NET.

W przypadku oceny programów na zawodach algorytmicznych zależy nam mimo wszystko na tym, żeby wykonanie programu jak najmniej odbiegało od wykonania na rzeczywistym procesorze. Z drugiej strony chcielibyśmy wyeliminować, lub chociaż ograniczyć, wymienione problemy. W tym momencie można się już domyślać, że pewnym rozwiązaniem może być stworzenie wirtualnej maszyny, która interpretuje rzeczywisty język maszynowy wybranego procesora (na przykład assembler architektury i686, czyli języka maszynowego procesorów Intel Pentium Pro). Odpowiednio skonstruowana maszyna wirtualna mogłaby również obliczać czas potrzebny na wykonanie poszczególnych instrukcji, zakładając pewien model wykonania kodu przez procesor. Zamiast jednak z przedwczesnym entuzjazmem ogłosić, iż jest to niezastąpione rozwiązanie wszystkich problemów, zwrócę uwagę na kilka rzeczy:

1. Stworzenie precyzyjnego modelu, który uwzględnia wszystkie aspekty działania współczesnych procesorów i zaimplementowane w nich elementy architektury, takie jak wielostopniowy potok przetwarzania instrukcji, pamięć podręczna, jednostka predykcji skoków i innych [8, 4], jest niemożliwe. Opisy sposobu działania wielu z tych mechanizmów są chronione tajemnicami handlowymi producentów procesorów. Z drugiej strony wymienione mechanizmy czasem znacząco wpływają na czas działania testowanego programu. Można oczywiście w eksperymentalny sposób badać sposób działania wymienionych komponentów i stworzyć mniej lub bardziej wierne modele ich działania, ale jest to kosztowne.

2. Program, który jedynie interpretowałby instrukcje języka maszynowego, już nawet nie emulując wymienionych powyżej komponentów, działa bardzo wolno.

   Spójrzmy na chwilę na bardzo prosty program pokazany na listingu 2.1.

Listing 2.1 Prosty program testowy w języku C

```
int main() {
    int i, j;
    for (i=0; i<100000000; i++, j++)
        j += 3;
    return 0;
}
```

Ten program skompilowany bez optymalizacji przy użyciu kompilatora GCC w wersji 4.3.2 kończy się na komputerze z procesorem 2.0 GHz w czasie 0,6 sek. Ten sam program uruchomiony pod emulatorem Bochs [1] działa 33,7 sek. Takie spowolnienie procesu sprawdzania zapewne byłoby nie do zaakceptowania na większości zawodów. Gdyby chcieć dodatkowo uwzględnić pominięte elementy procesora choć w przybliżony sposób, szybkość wirtualnej maszyny zmniejszyłaby się co najmniej kilkukrotnie.

Podsumowując powyższe rozważania, można wnioskować, że rozwiązaniem godnym uwagi nie jest dokładna symulacja zachowania się procesora, lecz stworzenie pewnego uproszczonego modelu, który pozwala oszacować czas działania programu przy użyciu miar, które można *efektywnie* wyznaczać.

2.2.1. Techniki niskopoziomowego pomiaru efektywności programów

Okazuje się, że wiele użytecznych miar można wyznaczyć nawet bez tworzenia pełnej maszyny wirtualnej emulującej wykonywanie kodu maszynowego. W dalszej części pracy opisuję techniki pozwalające na wyznaczanie wartości wybranych miar, przydatnych w modelowaniu czasu działania programów.

Sprzętowe liczniki zdarzeń w procesorach

Współczesne procesory posiadają sprzętowe podsystemy pozwalające na zliczanie ilości przeróżnych zdarzeń, jakie mają miejsce w procesorze. W szczególności można zliczać liczbę wykonanych instrukcji, liczbę cykli procesora spędzonych na czekaniu na dostęp do pamięci i wiele innych. Działanie tych liczników nie wpływa na wydajność uruchamianych programów.

[1] `http://bochs.sourceforge.net`

Zazwyczaj sposób obsługi rejestrów zliczających zdarzenia, jak i sam wybór zdarzeń, zależy od producentów i konkretnego modelu procesora. Mechanizm zliczania zdarzeń został wprowadzony przez Intela w serii procesorów Pentium. Początkowo można było otrzymać wartości następujących miar:

- liczba wykonanych odczytów/zapisów z/do pamięci podręcznej/operacyjnej,
- liczba instrukcji odczytanych z pamięci podręcznej/operacyjnej,
- liczba wykonanych instrukcji,
- liczba wykonanych instrukcji skoku,
- liczba cykli procesora, podczas których mają miejsce operacje na pamięci operacyjnej,
- liczba cykli procesora, podczas których nie jest wykonywany kod ze względu na oczekiwanie na operacje na pamięci operacyjnej,
- liczba wykonanych przerwań sprzętowych.

Wkrótce po tym również firma AMD wprowadziła możliwość zliczania zdarzeń, choć wprowadzono inny interfejs programistyczny. W kolejnych wersjach procesorów obydwu producentów wybór zdarzeń był wielokrotnie zmieniany, choć większość powyższych miar była dostępna lub zastąpiona bardzo podobnymi. Począwszy od serii procesorów Intel Core Solo, producent dokonał podziału monitorowanych zdarzeń na zdarzenia stałe dla architektury (*Architectural Performance Events*) oraz zależne od modelu procesora. Jednocześnie firma Intel zapewnia, że wszystkie zdarzenia należące do pierwszej grupy będą dostępne w przyszłych wydaniach procesorów. Oto miary, które możemy z nich uzyskać:

- liczba cykli zegara procesora, podczas których procesor nie był zatrzymany instrukcją `HLT`.
- liczba cykli zegara szyny danych, podczas których procesor nie był zatrzymany instrukcją `HLT`.
- liczba wykonanych instrukcji,
- liczba dostępów do pamięci podręcznej ostatniego poziomu,
- liczba dostępów do pamięci operacyjnej,
- liczba przetworzonych instrukcji skoku (niezależnie, czy skok warunkowy został wykonany czy nie),
- liczba przetworzonych instrukcji skoku warunkowego, w których jednostka predykcji niepoprawnie przewidziała spełnienie bądź niespełnienie warunku.

Szczegółowy opis wszystkich zliczanych zdarzeń można znaleźć w dokumentacji technicznej [9, 5, 3].

Na podstawie powyższego opisu można by wnioskować, że warto oprzeć tworzony model na zdarzeniach wypisanych w powyższej liście. Okazuje się jednak, że spośród miar dostępnych w szerokiej gamie procesorów, jedynie jedynie liczba wykonanych instrukcji nie zależy od konkretnego modelu procesora.

Instrumentacja kodu w czasie kompilacji

Innym podejściem do mierzenia wydajności programów jest *instrumentacja kodu*. W klasycznym podejściu polega ona na umieszczeniu w kodzie wykonywalnym programu dodatkowych instrukcji służących do zliczania zdarzeń bądź zapisywania innych informacji diagnostycznych. Takie podejście stosuje większość profilerów. Wymagają one wsparcia ze strony kompilatorów. Muszą one umożliwiać wstawianie w odpowiednie miejsca generowanego kodu niskopoziomowego dodatkowe instrukcje.

Przykładowo, rozważmy prostą pętlę, z którą już się spotkaliśmy przy demonstracji wydajności emulatora Bochs (listing 2.1, str. 16). Wyznaczymy, ile razy wykonywana jest każda z linijek powyższego programu. Do tego służy narzędzie do badania pokrycia kodu (*code coverage tool*). Aby to wykonać, można użyć następujących poleceń:

```
$ gcc -fprofile-arcs -ftest-coverage -o loop loop.c
$ ./loop
$ gcov ./loop
File 'loop.c'
Lines executed:100.00% of 4
loop.c:creating 'loop.c.gcov'
```

W wyniku tego powstał plik `loop.c.gcov`, w którym zapisano informacje, ile razy została wykonana każda linia programu:

```
        1:    1:int main() {
        -:    2:    int i, j;
100000001:    3:    for (i=0; i<100000000; i++, j++)
100000000:    4:        j += 3;
        1:    5:    return 0;
        -:    6:}
```

Przekazanie odpowiednich przełączników do kompilatora spowodowało umieszczenie w pliku wynikowym dodatkowych instrukcji zapisujących fakt wykonania poszczególnych linii. Porównanie kodu asemblerowego funkcji `main` bez oraz z instrumentacją można zobaczyć na listingu 2.2.

Nasuwa się więc pytanie: czy wykorzystanie instrumentacji sprawdziłoby się lepiej w praktyce? Można by zmodyfikować kompilator tak, aby umieszczał w kodzie dodatkowe instrukcje zbierające statystyki takie jak liczba i jakość wykonanych instrukcji czy dostępów do pamięci. Rozwiązanie to jednak ma zasadniczą wadę — modyfikowanie skomplikowanej aplikacji, jaką jest kompilator, która na dodatek podlega ciągłym zmianom, jest kosztowne zarówno w realizacji, jak i w późniejszym utrzymaniu. Co więcej, należało by zmodyfikować kompilatory wszystkich języków wykorzystywanych na zawodach.

Instrumentacja kodu w czasie wykonania

Kolejnym podejściem jest *instrumentacja kodu w czasie wykonania*. Polega ona na modyfikowaniu kodu działającego programu, aby osiągnąć efekt podobny do tego, co mógłby zrobić instrumentujący kompilator. W ten sposób powstały narzędzia, które działają niezależnie od wybranego języka programowania, a nawet nie wymagają do działania źródeł programu.

Po takim wstępie można oczekiwać, że na rynku dostępnych jest już wiele produktów wykorzystujących tę czy inną technikę do osiągnięcia różnych specyficznych celów. Pozwolę sobie opisać poniżej kilka takich, które wydają się być użyteczne do naszych zastosowań.

Listing 2.2 Wygenerowany kod w asemblerze bez instrumentacji (z lewej) oraz z instrumentacją (z prawej)

```
_main:
    pushl   %ebp
    movl    %esp, %ebp

    subl    $24, %esp




    movl    $0, -16(%ebp)
    jmp L2
L3:




    leal    -12(%ebp), %eax
    addl    $3, (%eax)
    leal    -16(%ebp), %eax
    incl    (%eax)
    leal    -12(%ebp), %eax
    incl    (%eax)
L2:
    cmpl    $99999999, -16(%ebp)
    jle L3




    movl    $0, %eax


    leave
    ret
```

```
_main:
    pushl   %ebp
    movl    %esp, %ebp
    pushl   %ebx
    subl    $20, %esp
    call    L6
L1$pb:
L6:
    popl    %ebx
    movl    $0, -16(%ebp)
    jmp L2
L3:
    leal    LPBX1-L1$pb(%ebx), %eax
    addl    $1, (%eax)
    adcl    $0, 4(%eax)
    leal    -12(%ebp), %eax
    addl    $3, (%eax)
    leal    -16(%ebp), %eax
    incl    (%eax)
    leal    -12(%ebp), %eax
    incl    (%eax)
L2:
    cmpl    $99999999, -16(%ebp)
    jle L3
    leal    LPBX1-L1$pb(%ebx), %eax
    leal    8(%eax), %eax
    addl    $1, (%eax)
    adcl    $0, 4(%eax)
    movl    $0, %eax
    addl    $20, %esp
    popl    %ebx
    leave
    ret
```

2.2.2. Narzędzia do analizy wydajności programów

Na początku chciałbym zwrócić uwagę na dwa najmniej „inwazyjne" narzędzia, czyli takie, które ani nie wymagają specjalnych usług systemowych, ani praw administratora. Jest to zestaw narzędzi pod nazwą Valgrind oraz biblioteka Pin. Poniżej przedstawiam pokrótce trzy produkty wykorzystujące tą technikę: narzędzia Valgrind, bibliotekę Pin oraz technologię LLVM-GCC. Na końcu wspomnę o bibliotece umożliwiającej wykorzystanie sprzętowych liczników zdarzeń — perfmon2.

Valgrind

Zacznijmy od Valgrinda [13], czyli pakietu narzędzi ułatwiających pracę programistów przez automatyczne diagnozowanie błędów w programach i analizy wydajności.

Niskopoziomowo Valgrind jest kompilatorem czasu rzeczywistego, tzn. stara się na bieżąco tłumaczyć napotkany kod maszynowy na kod instrumentowany. Valgrind nie wymaga specjalnej funkcjonalności ze strony kompilatora użytego do skompilowania analizowanego programu. Co więcej, potrafi badać skompilowane programy również wtedy, gdy ich kod źródłowy nie jest dostępny. Nie wymaga również żadnego konkretnego języka programowania.

W momencie, gdy uruchomiony program zaczyna wykonywać pewien kod po raz pierwszy, Valgrind przerywa jego działanie, analizuje ów kod — tłumaczy go na wewnętrzną reprezentację, która dopiero jest poddawana instrumentacji, a następnie z powrotem kompilowana do kodu maszynowego. Wewnętrzna reprezentacja jest niezależna od architektury procesora, na którym jest uruchamiany program, co umożliwia tworzenie uniwersalnych narzędzi działających również niezależnie od architektury procesora.

Niestety, z perspektywy kogoś, kto chciałby użyć Valgrinda do pomiaru efektywności działania programów na zawodach, dużą przeszkodą jest opisany sposób działania tego systemu. Umożliwia on wprawdzie dość dowolną instrumentację, jednak dopiero wewnętrznej reprezentacji, a nie oryginalnego kodu maszynowego. Nie zawsze ta reprezentacja pozwala dostatecznie dokładnie wyznaczyć miary wydajności oryginalnego programu. Tej wady nie posiada kolejny opisywany przeze mnie produkt — biblioteka Pin.

Pin

W 2004 roku światło dzienne ujrzał produkt firmy Intel pod nazwą Pin [12]. Jest to biblioteka instrumentacyjna, która została zaprojektowana z myślą o analizowaniu kodu przede wszystkim pod względem wydajności. Poza tym jest pod wieloma względami podobna do wspomnianego Valgrinda. Potrafi instrumentować kod programu podczas jego działania, jednak w przeciwieństwie do Valgrinda, nie przeprowadza dekompilacji kodu do wewnętrznej reprezentacji. Zamiast tego przepisuje kawałki kodu maszynowego przed pierwszym ich wykonaniem, dodając w odpowiednie miejsca elementy instrumentacji, oraz w minimalnym możliwym stopniu modyfikuje oryginalne wywołania.

Pin jest biblioteką. Aby otrzymać wyniki pomiarów wydajności programów, należy albo znaleźć narzędzie, które używa biblioteki Pin i mierzy to, czego potrzebujemy, albo stworzyć swoje. Programista, który chciałby skorzystać z biblioteki, ma do dyspozycji bardzo jasne i przejrzyste API oraz przystępnie napisaną dokumentację techniczną.

Spójrzmy na przykład zapożyczony z dokumentacji (listing 2.3). Jest to kompletna implementacja narzędzia, które zlicza liczbę instrukcji wykonanych przez testowany program. Warto zwrócić uwagę na funkcję `Instruction`, która została zarejestrowana w linii 40 jako funkcja, która będzie używana do instrumentacji pojedynczych instrukcji. Będzie ona wywołana przez bibliotekę Pin przy rekompilacji każdej instrukcji, co w efekcie spowoduje wstawienie w kompilowany kod odwołania do funkcji zliczającej `docount` przed każdą instrukcją maszynową oryginalnego programu.

Można oczekiwać, że tak zrealizowane zliczanie instrukcji — przez wstawienie wywołań funkcji zliczającej przed każdą instrukcję — dość negatywnie wpłynie na czas wykonania programu. Przykładowo, podany już wcześniej przykład prostej pętli (listing 2.1, str. 16), który na maszynie testowej działał 0,6 s., po uruchomieniu pod powyższą instrumentacją działa 2,7 s. Na szczęście biblioteka Pin umożliwia instrumentowanie kodu na różnych poziomach granularności, nie koniecznie pojedynczych instrukcji. W szczególności można napisać funkcję instrumentującą, która dla każdego podstawowego bloku kodu[2] jednokrotnie wykonuje

[2] *podstawowy blok kodu* — taki ciąg instrukcji, który jest wykonywany zawsze od początku do końca; żadna instrukcja w środku bloku podstawowego nie jest celem skoku, ani żadna instrukcja poza ostatnią nie może być instrukcją skoku.

Listing 2.3 Narzędzie do zliczania instrukcji przy użyciu biblioteki Pin

Źródło: zbiór przykładów dołączonych do biblioteki Pin

```
#include <iostream>
#include <fstream>
#include "pin.H"

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every instruction is executed
VOID docount() { icount++; }

// Pin calls this function every time a new instruction is encountered
VOID Instruction(INS ins, VOID *v)
{
    // Insert a call to docount before every instruction, no arguments are passed
    INS_InsertCall(ins, IPOINT_BEFORE, (AFUNPTR)docount, IARG_END);
}

KNOB<string> KnobOutputFile(KNOB_MODE_WRITEONCE, "pintool",
    "o", "inscount.out", "specify output file name");

// This function is called when the application exits
VOID Fini(INT32 code, VOID *v)
{
    // Write to a file since cout and cerr maybe closed by the application
    ofstream OutFile;
    OutFile.open(KnobOutputFile.Value().c_str());
    OutFile.setf(ios::showbase);
    OutFile << "Count " << icount << endl;
    OutFile.close();
}

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    // Initialize pin
    PIN_Init(argc, argv);

    // Register Instruction to be called to instrument instructions
    INS_AddInstrumentFunction(Instruction, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

Listing 2.4 Efektywniejsza wersja narzędzia do zliczania instrukcji (Pin)

Źródło: zbiór przykładów dołączonych do biblioteki Pin

```
#include <iostream>
#include <fstream>
#include "pin.H"

// The running count of instructions is kept here
// make it static to help the compiler optimize docount
static UINT64 icount = 0;

// This function is called before every block
VOID docount(INT32 c) { icount += c; }

// Pin calls this function every time a new basic block is encountered
// It inserts a call to docount
VOID Trace(TRACE trace, VOID *v)
{
    // Visit every basic block  in the trace
    for (BBL bbl = TRACE_BblHead(trace); BBL_Valid(bbl); bbl = BBL_Next(bbl))
    {
        // Insert a call to docount before every bbl, passing the number
        // of instructions
        BBL_InsertCall(bbl, IPOINT_BEFORE, (AFUNPTR)docount, IARG_UINT32,
                BBL_NumIns(bbl), IARG_END);
    }
}

// (Fini function omitted, it's the same as in the first example)

// argc, argv are the entire command line, including pin -t <toolname> -- ...
int main(int argc, char * argv[])
{
    // Initialize pin
    PIN_Init(argc, argv);

    // Register Instruction to be called to instrument instructions
    TRACE_AddInstrumentFunction(Trace, 0);

    // Register Fini to be called when the application exits
    PIN_AddFiniFunction(Fini, 0);

    // Start the program, never returns
    PIN_StartProgram();

    return 0;
}
```

funkcję zliczającą i przekazuje liczbę instrukcji w bloku jako argument. Takie rozwiązanie, przedstawione na listingu 2.4, dla tego samego programu działa 0,8 s.

LLVM GCC

Low Level Virtual Machine [11] to system przeznaczony w szczególności do tworzenia niezależnych od języka programowania narzędzi do analizy i optymalizacji kodu. LLVM definiuje własny asembler oraz dostarcza zestaw narzędzi, które pozwalają bądź wykonywać programy skompilowane do asemblera LLVM, bądź kompilować je do języka maszynowego wielu różnych procesorów. Asembler LLVM posiada niewiele prostych instrukcji i przez to nie jest ściśle związany z żadną architekturą, na której mógłby być wykonywany. Dołączone optymalizatory potrafią wykonywać optymalizacje na kodzie w tymże asemblerze. Następnie zestaw kompilatorów pozwala zoptymalizowany kod zamienić na kod maszynowy dla wybranej architektury, a nawet na poprawny program w języku C.

LLVM jest tak skonstruowany, że pozwala na względnie łatwą instrumentację kodu, co skutkuje tym, że nietrudno byłoby stworzyć aplikację, która dołączałaby do programów możliwość zliczania wykonanych instrukcji, np. tych w asemblerze LLVM, jak również innych parametrów takich jak dostępy do pamięci.

Niestety, choć kompilator GCC potrafi emitować kod w asemblerze LLVM, o tyle nie ma kompilatora Pascala, który by to potrafił. Dostosowanie języków posiadających wirtualną maszynę, takich jak Java, wymaga skompilowania tejże przy użyciu kompilatora emitującego kod LLVM. Niestety sama kompilacja nie pomaga, jeśli weźmiemy pod uwagę, że sama maszyna wirtualna Javy ma w sobie kompilator Just-in-Time [6], który nie potrafi emitować kodu LLVM. Z tego względu pozostawię ten system jedynie jako półstronicową wzmiankę w mojej pracy, a tymczasem przejdę do podsumowania.

Perfmon2

Perfmon2 [7] jest narzędziem pozwalającym na wykorzystanie sprzętowych liczników zdarzeń procesorów do analizy wydajności programów. Składa się z modułu jądra dla systemu Linux, oraz bibliotek, dzięki którym można tworzyć narzędzia kontrolujące testowane aplikacje.

Niewątpliwą zaletą tego rozwiązania jest to, że działa zarówno z procesorami firmy Intel, jak i AMD. Z drugiej strony wymaga użycia własnego modułu jądra, co utrudnia samodzielną instalację. Działa jedynie pod systemem Linux. Nie daje również możliwości instrumentowania kodu, a przez to ogranicza zakres dostępnych miar do jedynie tych, które raportuje procesor. Z tego względu nie zdecydowałem się na użycie tej biblioteki w eksperymentach.

Podsumowanie

Powyżej przedstawiłem szeroki wachlarz dostępnych produktów, które mniej lub bardziej pomagają w stworzeniu modelu procesora do naszych celów. Okazuje się, że istnieje wiele podejść szczególnie do realizacji prostego modelu, na którym chciałbym się skupić w swoich eksperymentach — zliczania instrukcji. W tabeli 2.2 podsumowałem cechy przedstawionych rozwiązań.

W pracy postanowiłem skupić się na dwóch rozwiązaniach — użyciu biblioteki Pin, oraz stworzeniu prostego systemu zliczającego instrukcje, który można w łatwy sposób zintegrować z istniejącą infrastrukturą oceniającą Olimpiady Informatycznej. Temu zagadnienie poświęcona jest kolejna część pracy.

	Valgrind	Pin	LLVM GCC	Perfmon2
Konieczność kompilacji kodu w specjalny sposób	NIE	NIE	TAK	NIE
Możliwość instrumentacji kodu napisanego w języku innym niż C/C++	TAK	TAK	NIE	TAK
Możliwość rozszerzenia ponad proste zliczanie instrukcji	TAK	TAK	TAK	[a]
Kompatybilność z systemem Windows	NIE	TAK	TAK	NIE
Zależność od konkretnego modelu procesora	TAK[b]	NIE	NIE	NIE[c]
Dobra dokumentacja techniczna	NIE	TAK	TAK	TAK
Dostępność na otwartej licencji	TAK[d]	TAK	TAK	TAK

[a] W bardzo ograniczony sposób, tzn. umożliwia jedynie zliczanie zdarzeń wspieranych przez procesor

[b] Działa jedynie na procesorach firmy Intel, choć to ograniczenie nie jest skutkiem przyjętych założeń, lecz konkretnej implementacji

[c] Procesory różnych producentów zliczają różne rodzaje miar, niewiele jest wspólnych

[d] Za wyjątkiem bibliotek do dezasemblacji kodu maszynowego, który jest opublikowany na licencji pozwalającej na dowolne wykorzystanie

Tablica 2.2 Porównanie dostępnych technologii pomocnych w implementacji modelu procesora.

2.3. Zliczanie instrukcji jako prosty model procesora

Celem niniejszej pracy jest ewaluacja techniki modelowania procesora w kontekście jej użycia do oceny rozwiązań zadań na konkursach algorytmicznych. Naturalnym następstwem osiągnięcia takiego celu jest wprowadzenie tej techniki na Olimpiadzie Informatycznej, bądź jej odrzucenie.

Do realizacji celu można podejść na dwa sposoby. Można najpierw stworzyć model, który według intuicji i najlepszej wiedzy autora wydaje się być wystarczający, wykonać eksperymenty, które potwierdzą lub odrzucą te przypuszczenia, a następnie model przyjąć lub kontynuować prace, albo odrzucić. Alternatywnie można stworzyć prosty model, dokładnie przeanalizować jego wady i zalety, i dopiero wtedy ocenić, czy warto tworzyć model bardziej zaawansowany. Jak najwcześniej można zainteresować środowisko Olimpiady, poddać wyniki wstępnych eksperymentów pod dyskusję w gronie członków Komitetu Głównego, starając uzyskać obraz pokazujący:

- jak wiernie stworzony model przybliża metodę używaną do tej pory,
- jakie jest nastawienie środowiska do wprowadzenia zmian, czy zalety nowego rozwiązania rekompensują jego wady,
- czy nowa miara jest bliższa temu, co chcemy zmierzyć.

Postanowiłem wybrać drugie z przedstawionych rozwiązań i na początek stworzyć prosty model i wszechstronnie go przeanalizować. Tym prostym modelem będzie *zliczanie instrukcji*.

Instrukcje, o których mowa, będą standardowymi instrukcjami dostępnymi w architekturze x86. Programy będą kompilowane w taki sam sposób, jak do tej pory. Aby móc prowadzić doświadczenia, w których wykorzystuję dotychczasową infrastrukturę oceniającą Olimpiady Informatycznej, przystosowałem istniejący System Internetowy Olimpiady do tego celu. Pozwolę sobie teraz opisać zmiany, które wprowadziłem.

2.4. Implementacja modułu zliczającego instrukcje dla Systemu Internetowego Olimpiady

System Internetowy Olimpiady (SIO) to oprogramowanie Olimpiady Informatycznej odpowiedzialne za automatyczne sprawdzanie rozwiązań przysyłanych na zawody. Składa się z interfejsu webowego, za pomocą którego odbywa się interakcja z zawodnikami — przysyłanie rozwiązań, sprawdzanie wyników, oglądanie rankingu itd. oraz modułu oceniającego. Do tej pory pomiar czasu działania programów odbywał się zawsze za pomocą mechanizmu systemu Linux, który raportuje, jaka ilość czasu procesora została wykorzystana na wykonywanie konkretnego procesu.

Aby przeprowadzić eksperymenty, zaimplementowałem w SIO zmiany, dzięki którym pomiar czasu działania ocenianych programów może się odbywać przez zliczanie wykonanych instrukcji. Przeprowadzenie tych zmian miało na celu przede wszystkim przeprowadzenie eksperymentów, więc starałem się zrobić to jak najmniej inwazyjnie. Zaproponowane rozwiązanie składa się z trzech części:

1. Modułu dla jądra systemu Linux o nazwie `exectime`. Jedyną jego funkcją jest zliczanie przy pomocy sprzętowych liczników zdarzeń (patrz punkt 2.2.1) liczby instrukcji wykonanych przez każdy proces w systemie.

Moduł jest kompatybilny zarówno z procesorami Intel, jak i AMD. Interfejsem modułu jest dodatkowy plik `/proc/<identyfikator procesu>/exectime`, z którego można odczytać jedną liczbę, czyli ilość instrukcji wykonanych przez dany proces. Liczone są jedynie instrukcje wykonane w trybie użytkownika.

2. Modyfikacji programu `supervisor`, będącego częścią SIO, odpowiedzialnego za wykonywanie programów zawodników w bezpiecznym środowisku oraz za odczytanie z systemu ilość zasobów zużytych przez nie. Modyfikacja polegała oczywiście na tym, żeby dodać możliwość odczytywania jako czasu działania programu ilości wykonanych instrukcji.

3. Modyfikacji wewnętrznej logiki SIO w celu dodania możliwości przełączania się pomiędzy dwoma sposobami pomiaru czasu.

Kod źródłowy stworzonego w tym celu modułu jądra, jak również zmodyfikowany program `supervisor`, są dostępne na stronie `http://www.mimuw.edu.pl/~accek/exectime/`.

It doesn't matter how beautiful your theory is,
it doesn't matter how smart you are.
If it doesn't agree with experiment, it's wrong.

Richard Feynman

Rozdział 3

Eksperymenty

Aby pokazać w sposób obiektywny wpływ różnych czynników oraz metod na wyniki pomiaru czasu działania programu, przeprowadziłem szereg eksperymentów. Najpierw chciałbym przedstawić analizę wpływu dwóch technologii stosowanych we współczesnych procesorach:

- pamięci podręcznej
- jednostki przewidywania skoków

na czas działania programów. W drugiej części tego rozdziału przedstawiam eksperyment polegający na przeprowadzeniu zawodów I stopnia ostatniej Olimpiady Informatycznej:

- po zmianie kompilatora,
- po zmianie maszyny sprawdzającej,
- po wykorzystaniu modelu, w którym jedynie mierzymy liczbę wykonanych instrukcji.

3.1. Środowisko eksperymentalne

Do przeprowadzenia eksperymentów użyłem czterech komputerów (w tym jednego „wirtualnego", tj. wykorzystującego model procesora). W tabeli 3.1 przedstawiam porównanie konfiguracji tych komputerów.

Nazwa	Procesor	Zegar procesora	Rozmiar cache'u	Rozmiar RAMu	Typ RAMu
Core 2 Duo	Intel Core 2 Duo T7100	1.80 GHz	2MB	2GB	DDR2 666 MHz
Xeon	Intel Xeon E5405	2.00 GHz	6MB	1GB	DDR2 666 MHz
Celeron	Intel Celeron CPU	3.06 GHz	256kB	512MB	DDR 333 MHz
Virtual	Procesor wirtualny	4.00 GHz[a]	—	—	—

[a]Procesor wirtualny potrafił wykonać dokładnie 4 mld. instrukcji na sekundę.

Tablica 3.1 Parametry komputerów użytych do eksperymentów.

Pozwolę sobie napisać parę słów o każdym z tych komputerów.

Core 2 Duo dobry laptop z 2007 r. Spośród komputerów testowych ten jest najbardziej zbliżony do „przeciętnego" komputera, na jakim może pracować wiele zawodników Olimpiady Informatycznej.

Xeon maszyna serwerowa wiodącego producenta. Model ten jest reklamowany jako „uniwersalny serwer”. Podobne serwery są wykorzystywane na szeroką skalę w przemyśle IT. Rok produkcji: 2008.

Celeron komputer stacjonarny oparty o najtańsze w momencie zakupu dostępne podzespoły. *Aktualnie używany do oceny rozwiązań na Olimpiadzie Informatycznej.* Rok produkcji: 2006.

Virtual nie jest rzeczywistym komputerem, lecz odpowiada modelowi procesora, który potrafi wykonać $4 \cdot 10^9$ instrukcji na sekundę. Przez instrukcję rozumiem tutaj pojedyncze polecenie asemblera platformy i686[1].

Eksperymenty rozpocząłem od doświadczenia polegającego na sprawdzeniu, ile najprostszych możliwych instrukcji są w stanie wykonać poszczególne procesory w ciągu sekundy. W tym celu napisałem program, który w pętli wykonuje instrukcje `NOP` (listing 3.5). W jednym przebiegu wykonuje się ich 100, aby logika pętli zajmowała możliwie mało czasu.

Listing 3.5 Program użyty do wyznaczenia liczby instrukcji, które procesor jest w stanie wykonać w ciągu sekundy.

```
int main() {
  register int i;
  for (i=0; i<100000000; i++) {
    asm volatile (
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
        "nop;nop;nop;nop;nop;nop;nop;nop;nop;nop;"
    );
  }
  return 0;
}
```

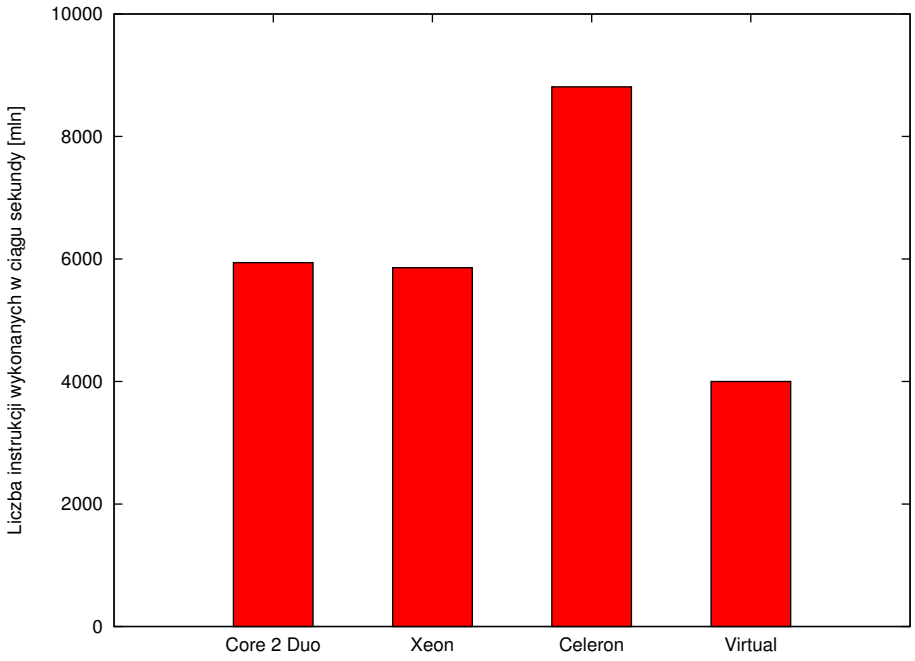
Powyższy program został wielokrotnie uruchomiony na komputerach testowych. Uśrednione wyniki przedstawiono na rys. 3.1.

Zadziwiające jest to, że komputer Core 2 Duo, taktowany zegarem 1.8 GHz potrafi wykonać w ciągu sekundy aż 6 mld. instrukcji, czyli ponad 3 na każdy cykl zegara. Pozostałe procesory również potrafią wykonywać więcej niż jedną instrukcję na cykl.

Podobne porównanie wykonałem odnoście efektywności dostępów do pamięci. Program, który dziesięciokrotnie wypełnia tablicę o rozmiarze 256MB za pomocą funkcji `memset` został uruchomiony na maszynach testowych. Wyniki przedstawiono na rysunku 3.2.

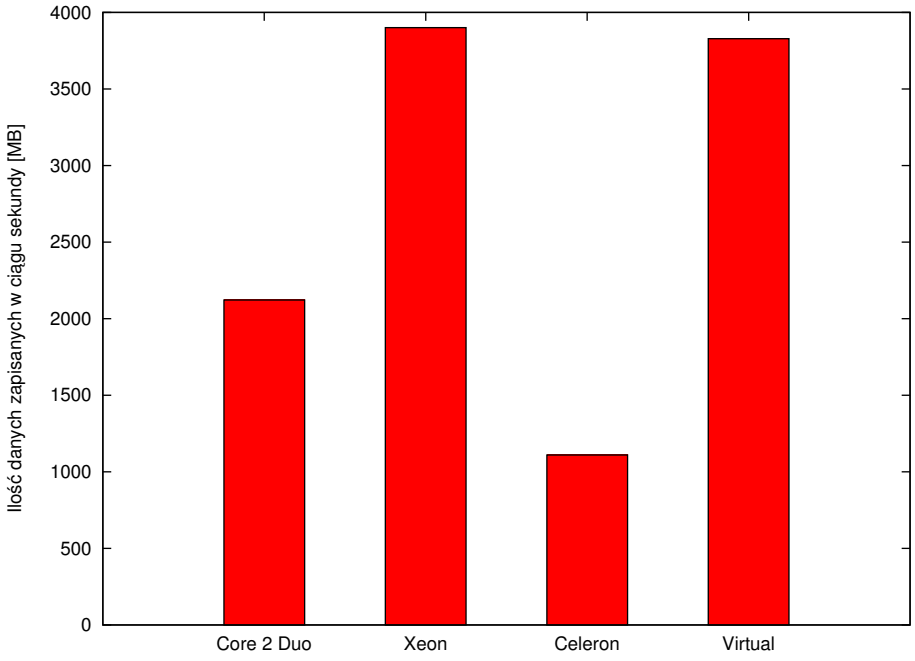
Pewnym zaskoczeniem jest prawie dwukrotnie większa szybkość zapisu na komputerze Xeon w porównaniu z Core 2 Duo mimo, że oba komputery posiadają pamięć DDR2 666 MHz.

[1]Instrukcje z prefiksem `REP` itp. są liczone wielokrotnie.



Rysunek 3.1 Liczba instrukcji `NOP` wykonywanych przez różne procesory w ciągu sekundy.

Ich procesory są również taktowane podobnymi zegarami. Podejrzewam, że różnica wynika z różnego sposobu połączenia kości pamięci w maszynach serwerowych i laptopach.



Rysunek 3.2 Szybkość zapisów do pamięci na testowanych komputerach.

3.2. Znaczenie pamięci podręcznej procesora

Po krótkim porównaniu maszyn, czas przejść do bardziej obrazowych eksperymentów. W tym punkcie przedstawiam wyniki doświadczeń pokazujących jak bardzo procesory różnią się szybkością wykonywania takich samych programów. Mogą nam one pomóc w odpowiedzi na takie oto pytania:

> Który procesor najlepiej nadaje się do oceny zadań
> na Olimpiadzie Informatycznej?
>
> Jaki wpływ na czas działania programów ma zmiana procesora?

Eksperymenty zostały przygotowane, by pokazać różnice spowodowane przez pamięć podręczną procesora.

Użyty kompilator

Programy użyte do eksperymentów prezentowanych w tym punkcie zostały skompilowane za pomocą kompilatora *gcc* w wersji 4.3.2 (dla architektury i686) poleceniem

```
gcc -Os -static -o plik_wykonywalny plik_źródłowy
```

Opcja `-Os` powoduje włączenie tych samych optymalizacji, co często używana opcja `-O2`, za wyjątkiem tych, które polegają na umieszczaniu w kodzie dodatkowych instrukcji `NOP`, aby wyrównać adresów celów skoków do wielokrotności 16 bajtów. W ten sposób wczytywanie kodu do wykonania w wyniku skoków może być efektywniejsze. W przypadku krótkich programów, które w całości mieszczą się w pamięci podręcznej instrukcji, optymalizacja ta nie jest potrzebna, a wręcz może spowolnić wykonanie programu. Mniejsze znaczenie ma to w przypadku rzeczywistych procesorów, które potrafią dekodować kilka instrukcji w jednym cyklu i ignorować instrukcje `NOP` we wczesnej fazie ich przetwarzania, natomiast w przypadku wirtualnego procesora instrukcje `NOP`, jak wszystkie inne instrukcje, trwają dokładnie jeden cykl.

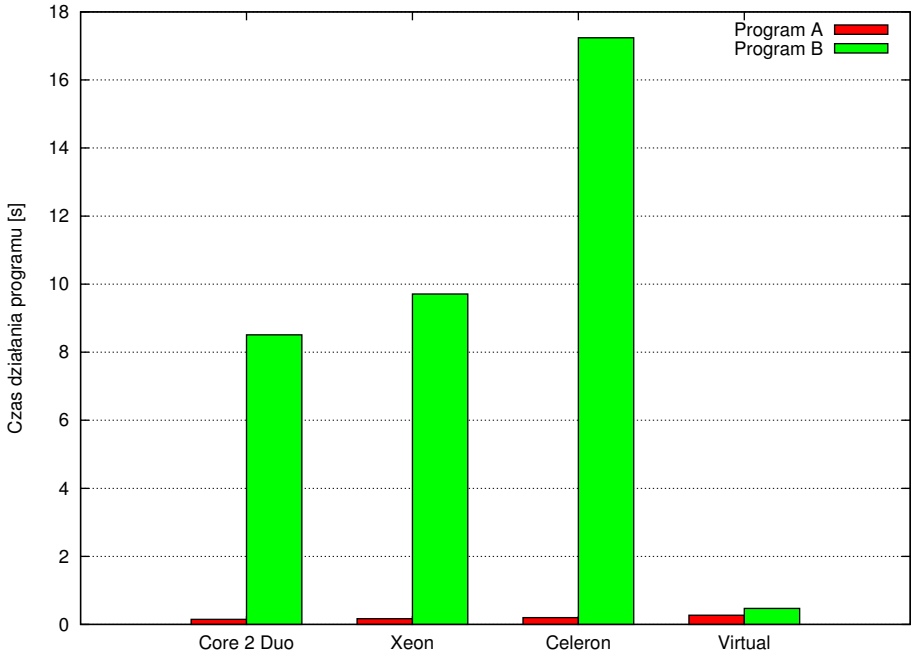
Czym jest pamięć podręczna?

Pamięć podręczna procesora (cache) służy do tymczasowego przechowywania danych pobranych z pamięci operacyjnej oraz tych, które czekają na zapis. Dostępy do pamięci podręcznej są wielokrotnie szybsze niż dostępy do pamięci operacyjnej, a więc każda operacja, która może wykorzystać cache zajmuje mniej czasu. Cache jest w większości przeźroczysty dla działających programów, tzn. nie wymaga się umieszczania w kodzie programu instrukcji sterujących cachem (są wyjątki od tej reguły, ale na potrzeby tej pracy możemy je pominąć). Mimo to okazuje się, że w niektórych sytuacja warto wiedzieć o jego istnieniu, gdy chcemy, aby nasze programy działały szybko.

Jedną z prostych zasad jest ta, która mówi, iż warto wykonywać dostępy do pamięci w sposób liniowy, czyli wykorzystując kolejne adresy, w przeciwieństwie do „losowego skakania tu i ówdzie". Jest tak, ponieważ procesor pobiera z pamięci pełne *linie cache'u*, czyli spójne kawałki pamięci o pewnej ustalonej długości, zazwyczaj 64 lub 128 bajtów. Jeśli kolejne wykonywane przez nas operacje wykorzystują po kilka bajtów w odległych miejscach, to procesor niepotrzebnie będzie pobierał pełne np. 64-bajtowe obszary pamięci.

Eksperymenty

Żeby sprawdzić, jakie znaczenie ma ta zasada, przygotowano dwa programy. Oba wypełniającą tablicę znaków wielkości 256 MB. Jeden z nich zapisuje w pętli bajty o kolejnych adresach, natomiast drugi wykonuje to samo skokami co 128 B (listingi 3.6 oraz 3.7). Rysunek 3.3 przedstawia porównanie czasów działania tych dwóch programów na testowych komputerach.



	Program A	Program B
Core 2 Duo	0.15	8.51
Xeon	0.17	9.71
Celeron	0.20	17.24
Virtual	0.27	0.47

Rysunek 3.3 Porównanie wydajności liniowych i nieliniowych dostępów do pamięci.

Natychmiast rzuca się w oczy następujący wniosek:

> Nieliniowe dostępy do pamięci mogą spowolnić program **ponadsześćdziesięciokrotnie** w porównaniu z podobnym programem, który wykonuje dostępy do pamięci liniowo.

Naturalne jest, że tego zjawiska nie zauważamy na komputerze z wirtualnym procesorem, który po prostu wykonuje 4 mld instrukcji na sekundę, niezależnie od tego, czy są to dostępy do pamięci, czy nie. Natomiast na pytanie „Dlaczego na wirtualnym procesorze drugi program działa wolniej?" odpowiedź brzmi „Ze względu na dodatkową operację `(i*128) % sizeof(c)`, wykonywaną w każdym przebiegu pętli".

Powyższy przykład, jakkolwiek dający czasem do myślenia, wydaje się być dość odległy od praktyki. Któż w tak dziwaczny sposób przegląda tablice? Niestety można skonstruować

Listing 3.6 Program A — wypełnia kolejne bajty pamięci.

```
char c[1<<28];  // 256MB

int main() {
  int i;
  for (i=0; i<sizeof(c); i++) {
    c[i] = i;
  }
  return 0;
}
```

Listing 3.7 Program B — wypełnia pamięć skokami po 128B.

```
char c[1<<28];  // 256MB

int main() {
  int i;
  for (i=0; i<sizeof(c); i++) {
    c[(i*128) % sizeof(c)] = i;
  }
  return 0;
}
```

bardziej życiowy przykład, który chciałbym teraz przedstawić.

Warto przy tym wiedzieć, że w języku C (jak również w C++, Pascalu i wielu innych) tablica dwuwymiarowa jest realizowana jako jeden spójny obszar pamięci na tyle duży, żeby zmieścić wszystkie komórki tablicy. Komórki tablicy są umieszczone obok siebie w takim obszarze w następującej kolejności (dla tablicy dwuwymiarowej $m \times n$):

`t[0][0]`, `t[0][1]`, ..., `t[0][n-1]`,
`t[1][0]`, `t[1][1]`, ..., `t[1][n-1]`,
...,
`t[m-1][0]`, `t[m-1][1]`, ..., `t[m-1][n-1]`.

Pamiętając o tym, rozważmy trzy programy, które wypełniają dwuwymiarową tablicę mającą 10^8 komórek.

A. Pierwszy przegląda tablicę bajtów o rozmiarze 10 000 × 10 000 wierszami[2] i przez to wykonuje liniowe dostępy do pamięci (listing 3.8).

B. Drugi przegląda taką samą tablicę kolumnami. Kolejno przeglądane wartości są zatem oddalone o 10 000 komórek pamięci (listing 3.9).

C. Trzeci przegląda tablicę 1 000 000 × 100 kolumnami (listing 3.10).

Listing 3.8 Program A

Tablica 10 000 × 10 000 przeglądana wierszami.

```
char a[10000][10000];

int main() {
  int i, j;
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      a[i][j] = i;
  return 0;
}
```

Listing 3.9 Program B

Tablica 10 000 × 10 000 przeglądana kolumnami.

```
char a[10000][10000];

int main() {
  int i, j;
  for (i=0; i<10000; i++)
    for (j=0; j<10000; j++)
      a[j][i] = i;
  return 0;
}
```

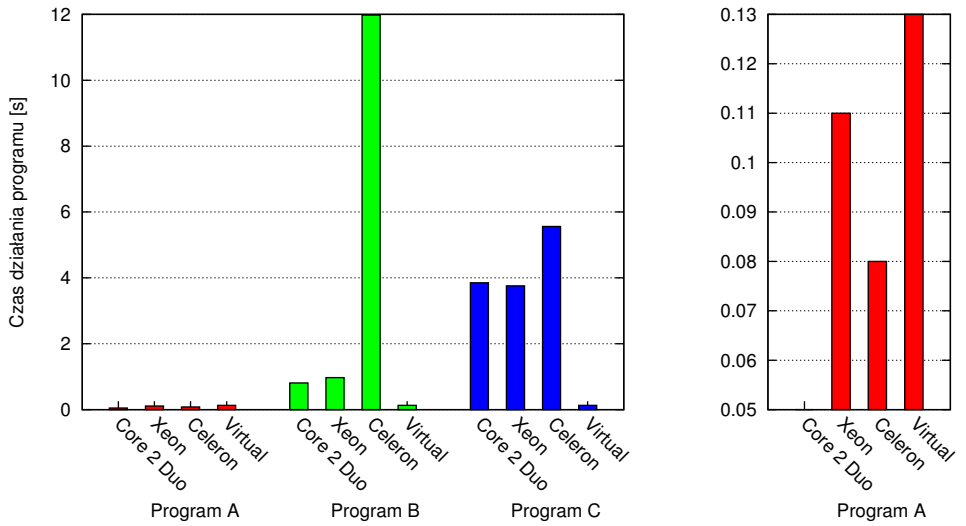
Listing 3.10 Program C

Tablica 1 000 000 × 100 przeglądana kolumnami.

```
char a[1000000][100];

int main() {
  int i, j;
  for (i=0; i<100; i++)
    for (j=0; j<1000000; j++)
      a[j][i] = i;
  return 0;
}
```

[2]Napis `t[i][j]` oznacza i-ty wiersz oraz j-tą kolumnę tablicy.



	A	B	C
Tablica:	10000 x 10000	10000 x 10000	1000000 x 100
Kierunek przeglądania:	kolumnami	wierszami	wierszami
Core 2 Duo	0.05	0.81	3.85
Xeon	0.11	0.97	3.76
Celeron	0.08	11.98	5.56
Virtual	0.13	0.13	0.13

Rysunek 3.4 Porównanie wydajności programów wypełniających tablicę dwuwymiarową.

Spójrzmy na czas wykonania tych programów na komputerach testowych (rys. 3.4). Znów widzimy, że liniowe dostępy do pamięci są wielokrotnie szybsze. Można się zastanawiać, skąd się wzięła różnica pomiędzy programami B i C. W końcu obydwa przechodzą po tablicy w „złej" kolejności. Dzieje się tak, ponieważ podczas przeglądania drugiej kolumny tablicy, cześć z „jedynie" 10 000 elementów może być wciąż w cache'u procesora, bo podczas przejścia pierwszej kolumny procesor pobierał dane z pamięci całymi liniami cache'u[3]. Podobnie przy przeglądaniu kolejnych kolumn. Jeśli tablica ma 1 000 000 wierszy, szansa na napotkanie w cache'u pozostałości z pierwszego przebiegu jest dużo mniejsza.

Na podsumowanie powyższego eksperymentu pozwolę sobie wyróżnić trzy obserwacje:

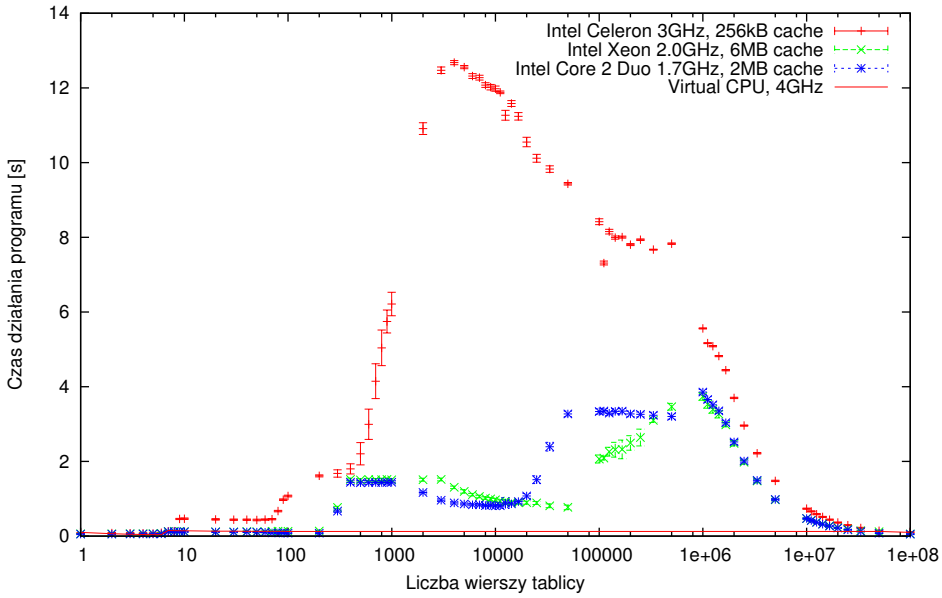
Przeglądanie kolumnami dwuwymiarowej tablicy może być **ponaddwudziestokrotnie** wolniejsze od przeglądania tej samej tablicy wierszami.

Przeglądanie kolumnami dwuwymiarowej tablicy o rozmiarze $1\,000\,000 \times 100$ może być **ponadsześćdziesięciokrotnie** wolniejsze od przeglądania wierszami kwadratowej tablicy o tej samej wielkości.

[3]Patrz punkt 3.2, str. 30, gdzie pokrótce opisana jest zasada działania cache'u procesora.

Przeglądanie kolumnami dwuwymiarowej tablicy o rozmiarze $1\,000\,000 \times 100$ może być **kilkukrotnie wolniejsze bądź kilkukrotnie szybsze** od przeglądania kolumnami kwadratowej tablicy o tej samej wielkości, w zależności od komputera, na którym testujemy.

Idąc dalej tą drogą, wykonano szereg eksperymentów z różnymi wielkościami tablic, aby przekonać się, jak poszczególne procesory reagują na różne wzorce dostępu do pamięci. W każdym z eksperymentów program przechodził po dwuwymiarowej tablicy kolumnami. Tablica miała zawsze 10^8 elementów, natomiast jej wymiary były zmienne. Czasy działania przedstawiono na rys. 3.5. Zaznaczono również odchylenia standardowe pomiarów.

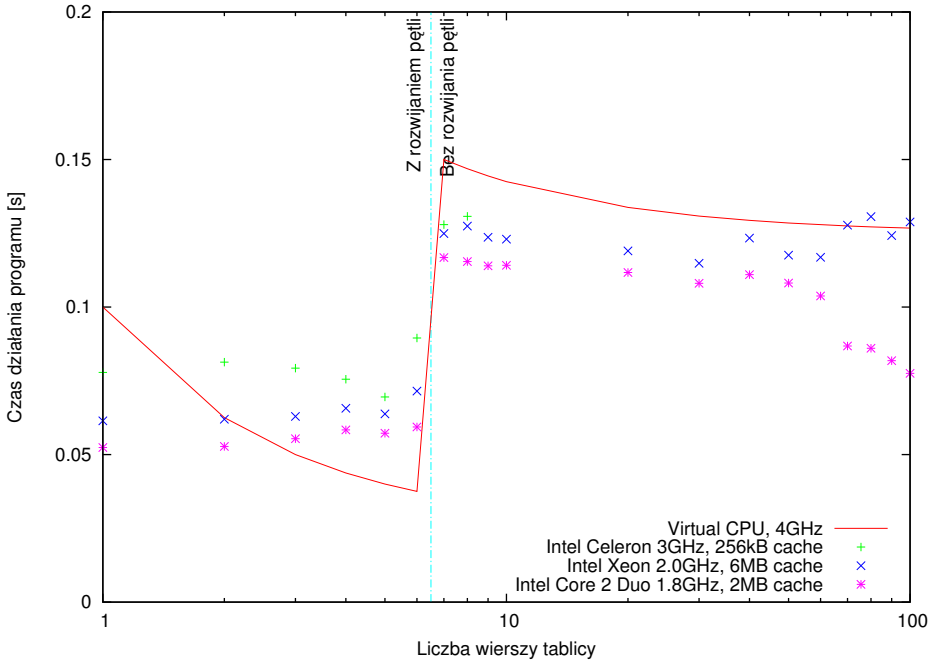


Rysunek 3.5 Wydajność przeglądania 10^8-elementowych tablic różnych wymiarów.

Widzimy, że kształty poszczególnych krzywych nie mają ze sobą wiele wspólnego, może poza ogólnym trendem wskazującym, że im więcej cache'u, tym lepiej. A i tak pomiędzy 1 000 a 10 000 wierszy tak nie jest. Również same kształty ciężko opisać prostymi regułami. Ta demonstracja pokazuje, że na wydajność takiej operacji jak przechodzenie po tablicy kolumnami ogromny wpływ ma użycie konkretnego procesora. Nawet wiedząc, jaki mamy procesor, ciężko ocenić, jak zmienia się wydajność programu w zależności od wymiarów tablicy. Na wirtualnym procesorze oczywiście nie widać prawie żadnych wahań wydajności. O takim zachowaniu można myśleć w innych kategoriach, a mianowicie

Procesor wirtualny zachowuje się podobnie do procesora, który ma **nieskończenie dużo cache'u**.

Warto jeszcze przyjrzeć się wynikom dla tablic o niewielkiej liczbie wierszy. Na powyższym wykresie niewiele widać w tym rejonie, dlatego spójrzmy na powiększenie tego fragmentu (rys. 3.6).



Rysunek 3.6 Wydajność przeglądania 10^8-elementowych tablic różnych wymiarów (fragment wykresu dla tablic do stu wierszy).

Możemy zaobserwować dwa zjawiska:

1. Po pierwsze czas działania programu dla jednowierszowej tablicy jest większy niż dla dwu-, trzy-i tak dalej, dla wirtualnego komputera. To zjawisko można uzasadnić większym narzutem na wielokrotną inicjację wewnętrznej pętli algorytmu. Istotnie dla jednowierszowej tablicy wewnętrzna pętla przebiega 10^8 razy, dla dwuwierszowej dwa razy mniej i tak dalej.

   Tym niemniej pozostaje pytanie, dlaczego nie widać tego dla pozostałych procesorów. Najprawdopodobniej wynika to ze specyfiki ich sprzętowej architektury — współczesne procesory potrafią wewnętrznie zrównoleglić wykonywanie operacji. To może powodować, że nowsze procesory potrafią zniwelować narzut na wykonanie wewnętrznej pętli przez zrównoleglenie wykonania logiki pętli oraz dostępów do pamięci w ten sposób, że operacje pamięciowe są dominujące.

2. Drugim zjawiskiem jest gwałtowny spadek wydajności przy przechodzeniu z przeglądania tablicy sześciowierszowej do siedmiowierszowej i to niezależnie od procesora. Tym razem okazuje się, że „winowajcą" jest kompilator, a konkretnie jego optymalizator. W sytuacjach, gdy wewnętrzna pętla wykonuje nie więcej niż 6 przebiegów, kompilator postanowił ją rozwinąć, czyli zastąpić wielokrotnym powtórzeniem instrukcji wewnątrz pętli [2]. Widać to na załączonych listingach (3.12 oraz 3.11).

Powyższe wyniki pokazują, że w specyficznych sytuacjach cache procesora może mieć kolosalny wpływ na czas działania programów. I choć programy na zawodach algorytmicznych rzadko są tak proste jak te testowane powyżej, to jednak przeglądanie tablicy dwuwymiarowej często pojawia się jako element rozwiązania. Warto zwrócić uwagę, że przechodzenie dwuwymiarowej tablicy nie jest jedyną operacją tak wrażliwą na cache.

Listing 3.11 Kod asemblerowy pętli wypełniającej tablicę o sześciu wierszach.

```
    xorl    %eax, %eax
.L2:
    movb    %al, a(%eax)
    movb    %al, a+16666666(%eax)
    movb    %al, a+33333332(%eax)
    movb    %al, a+49999998(%eax)
    movb    %al, a+66666664(%eax)
    movb    %al, a+83333330(%eax)
    incl    %eax
    cmpl    $16666666, %eax
    jne .L2
```

Listing 3.12 Kod asemblerowy pętli wypełniającej tablicę o siedmiu wierszach.

```
    xorl    %edx, %edx
    jmp .L2
.L3:
    movb    %dl, (%eax)
    addl    $14285714, %eax
    cmpl    %ecx, %eax
    jne .L3
    incl    %edx
    cmpl    $14285714, %edx
    je  .L4
.L2:
    leal    a(%edx), %eax
    leal    a+99999998(%edx), %ecx
    jmp .L3
.L4:
```

Okazuje się bowiem, że z podobną sytuacją mieliśmy do czynienia w ubiegłej Olimpiadzie Informatycznej w przypadku jedynego rozwiązania wzorcowego w zadaniu „Przyspieszenie algorytmu”, co nie zostało do tej pory zauważone. W efekcie nieświadomie przyszło nam ustawić limity czasowe dla tego zadania w ten sposób, że kilka programów o gorszej złożoności niż oczekiwana, otrzymało 100 punktów w tym zadaniu. Więcej informacji znajduje się w punkcie 3.3.8 poświęconym temu właśnie zadaniu.

3.3. Wykorzystanie modelu procesora na Olimpiadzie Informatycznej

Powyżej przedstawiłem wyniki eksperymentów pokazujących wpływ jednego mechanizmu — cache'u procesora — na czas działania programów. Teraz chciałbym omówić doświadczenia bezpośrednio związane z zasadami oceny na Olimpiadzie Informatycznej i podobnych konkursach algorytmicznych.

Eksperyment polegał na ocenieniu rozwiązań przysłanych na **I etap XVI Olimpiady Informatycznej** w trzech środowiskach:

1. W oficjalnym środowisku zgodnym z dokumentem „Ustalenia techniczna na I etap XVI OI" oraz przy użyciu tych samych maszyn sprawdzających, które zostały użyte na olimpiadzie.

2. W środowisku ze zmienioną wersją kompilatora dla języków C i C++, lecz wciąż na tych samych maszynach sprawdzających (Celeron, konfiguracja przedstawiona w tabeli 3.1). Kompilator GCC zmieniono z wersji **4.1.1** do **4.1.3**.

3. W środowisku z kompilatorem GCC 4.1.3 na maszynie Xeon (tabela 3.1).

4. W środowisku z kompilatorem GCC 4.1.3 na maszynie z wirtualnym procesorem (pozycja Virtual w tabeli 3.1).

W eksperymentach nie uwzględniono rozwiązań w Javie ze względu na ich niewielką liczbę (patrz tabela 3.2) oraz brak możliwości zliczania instrukcji dla programów w Javie przy użyciu aktualnego systemu. Zwracam też uwagę na to, że w roku szkolnym 2009/2010 olimpiada zrezygnowała z wsparcia dla Javy.

W każdej z wymienionych konfiguracji ponownie ustawiono limity czasowe zgodnie z opisami zawartymi w dokumentach opracowań zadań. Następnie ponownie oceniono wszystkie ostateczne zgłoszenia przysłane przez zawodników oraz wszystkie programy przygotowane w procesie opracowania zadania.

Język programowania	Liczba zgłoszeń
C++	3543
Pascal	489
C	138
Java	44

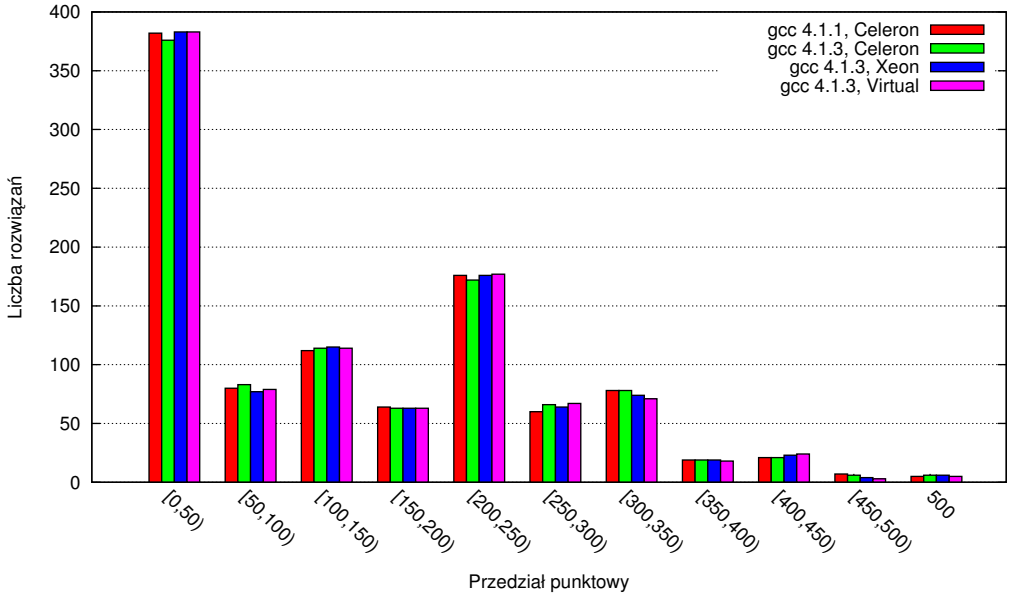
Tablica 3.2 Liczba programów w poszczególnych językach programowania ocenionych na I etapie XVI OI.

Zebrane dane przeanalizowano osobno dla każdego zadania, a także podsumowano dla całej rundy. Na początku chciałbym przedstawić sumaryczne dane, a potem przejrzeć się charakterystyce wyników dla poszczególnych zadań.

3.3.1. Rozkład punktów

Wykresem, od którego chciałbym zacząć jest histogram przedstawiający rozkład liczby punktów uzyskanych przez zawodników na I etapie (rys. 3.7). Widzimy, że sumaryczne wyniki są podobne we wszystkich środowiskach. Rezultaty oceniania z użyciem modelu procesora (kolor

purpurowy) są najbardziej zbliżone do wyników na maszynie Xeon. Zgadza się to z wnioskiem zawartym w poprzednim punkcie, mówiącym, że wirtualny procesor zachowuje się tak, jakby posiadał bardzo dużo pamięci podręcznej.



Rysunek 3.7 Rozkład punktów w I etapie XVI OI

3.3.2. Porównanie wyników poszczególnych zawodników

Oczywiście na podstawie samego histogramu nie należy wyciągać wniosków o przydatności proponowanej metody w praktyce. Może się bowiem okazać, że ogólny rozkład wyników zmienił się nieznacznie, lecz pewne klasy rozwiązań mogą być nieoczekiwanie faworyzowane. Aby bliżej przyjrzeć się różnicom wyników dla poszczególnych zawodników, spójrzmy na rysunki na następnych stronach.

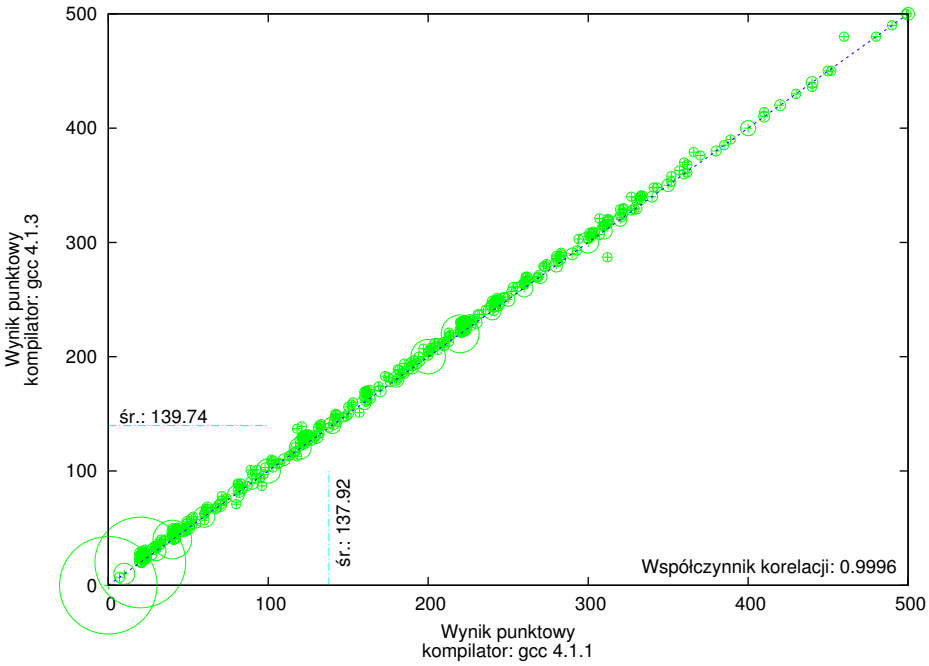
Każda strona jest poświęcona innemu porównaniu:

- str. 39 — kompilator gcc 4.1.1 vs gcc 4.1.3,
- str. 40 — komputer Celeron[4] vs komputer Xeon,
- str. 41 — komputer Celeron vs model procesora

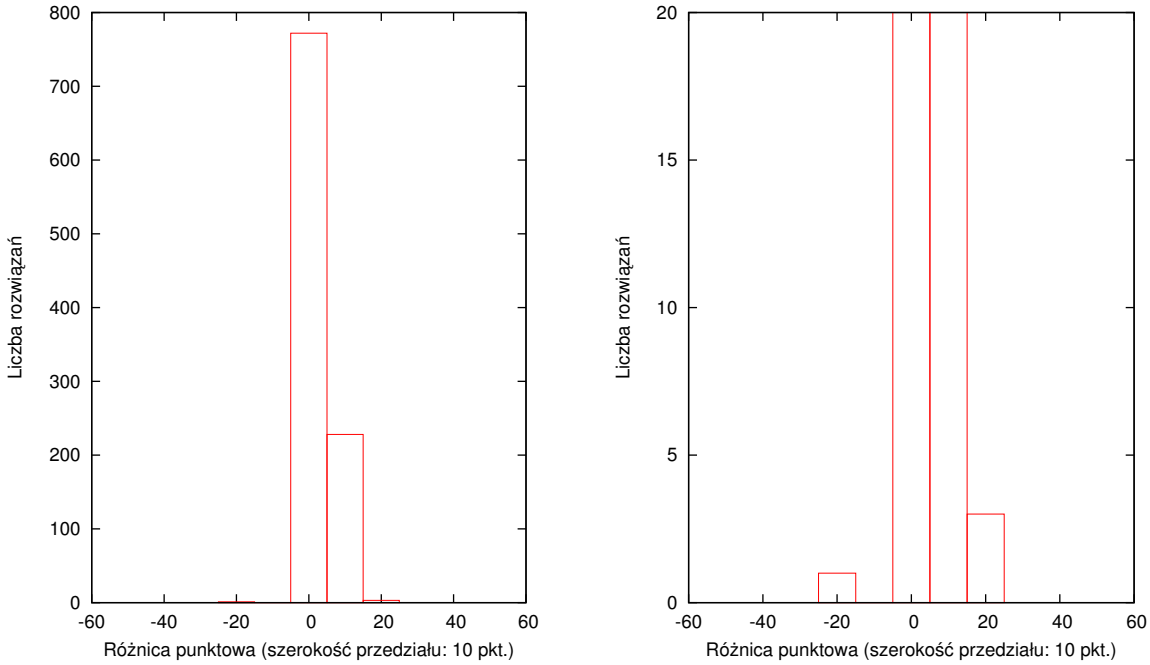
Górny wykres przedstawia porównanie wyników poszczególnych zawodników. Na osi X umieszczono wynik punktowy zawodnika przy ocenie w jednym z porównywanych środowisk. Na osi Y jest wynik zawodnika w drugim środowisku oceny. Każdemu zawodnikowi odpowiada jeden punkt wykresu. W przypadku gdy punkty odpowiadające kilku zawodnikom miałby leżeć w dokładnie tym samym miejscu, narysowano jeden proporcjonalnie większy punkt.

Na dole strony znajdują się dwa histogramy przedstawiające rozkład zmian wyników zawodników. Na osi X znajdują się kolejne przedziały punktowe, a na osi Y liczba zawodników, których wynik zmienił się o wybraną liczbę punktów. Oba histogramy przedstawiają to samo, różnią się jedynie skalą na osi Y.

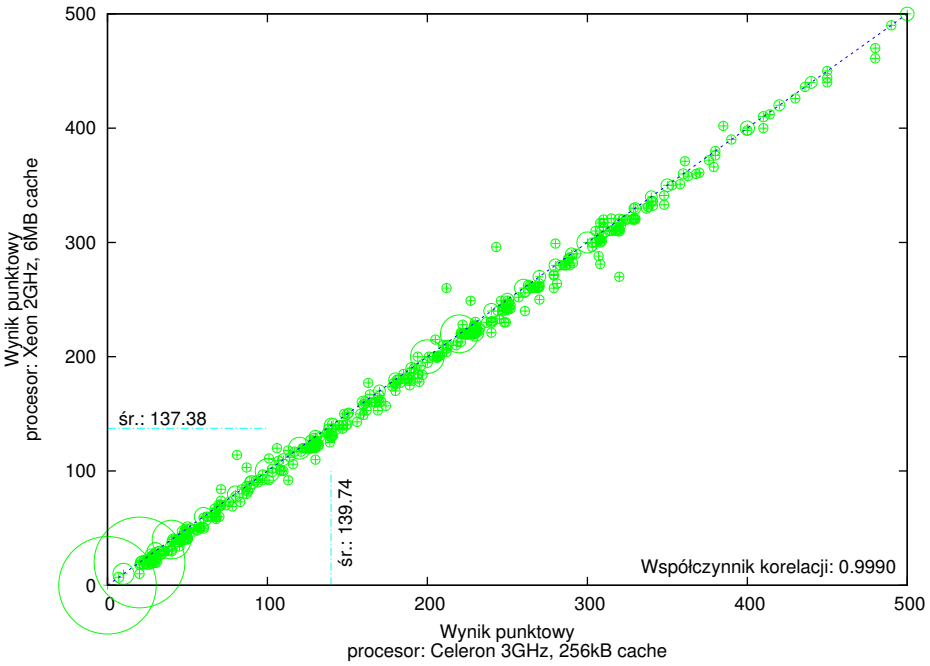
[4]Celeron to maszyna aktualnie używana do sprawdzania zadań na Olimpiadzie Informatycznej.



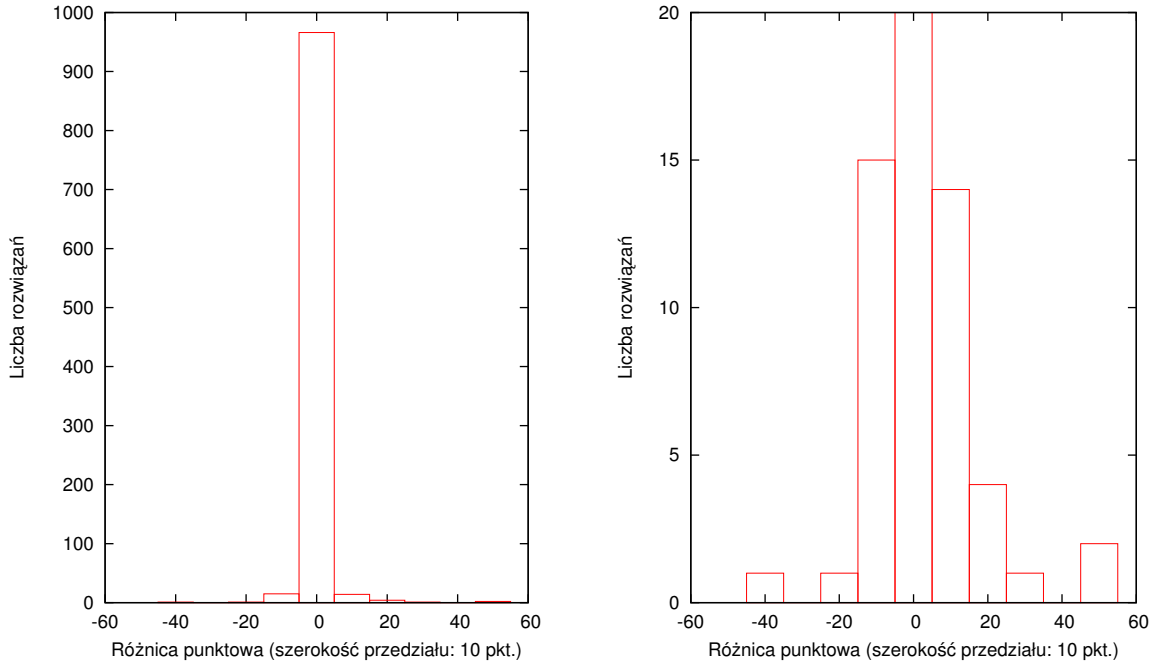
Rysunek 3.8 Porównanie wyników zawodników przy ocenie kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron.



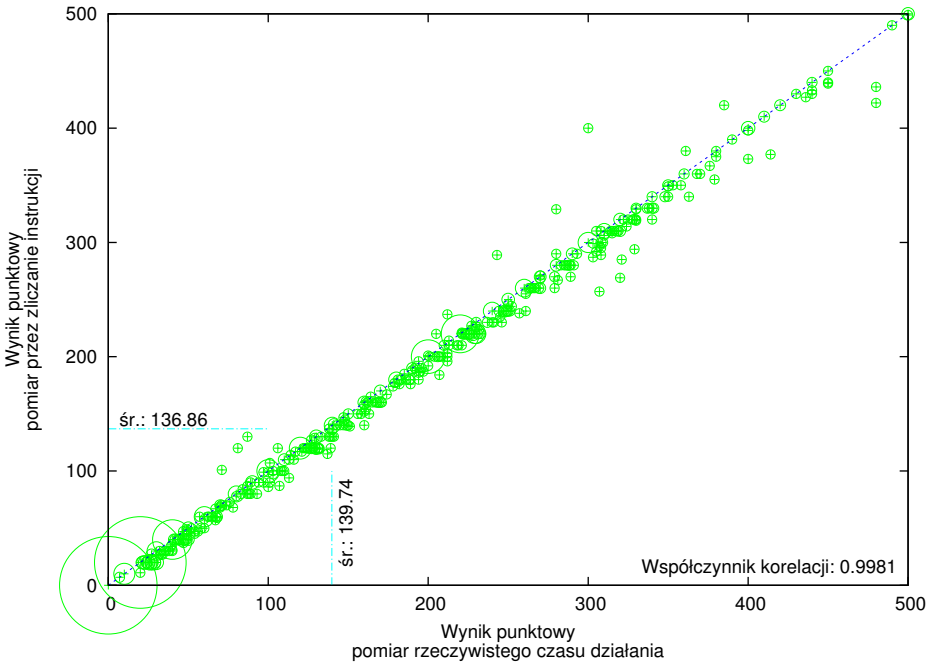
Rysunek 3.9 Rozkład różnic wyników zawodników przy ocenie kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron.



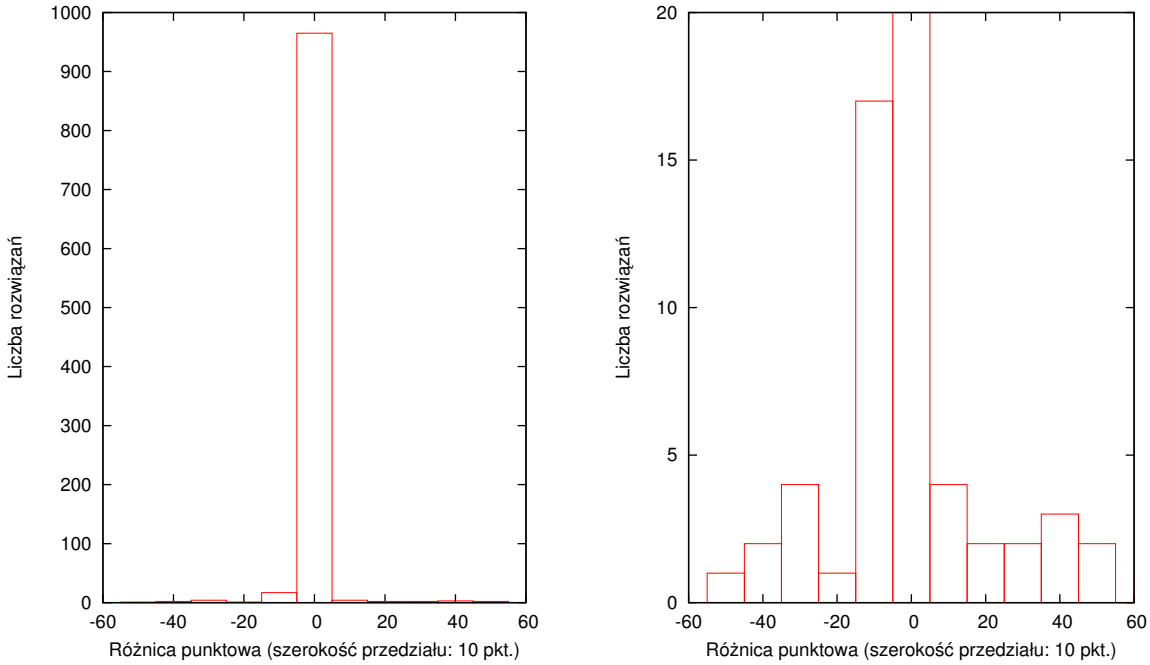
Rysunek 3.10 Porównanie wyników zawodników przy ocenie kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon.



Rysunek 3.11 Rozkład różnic wyników zawodników przy ocenie kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon.



Rysunek 3.12 Porównanie wyników zawodników przy ocenie kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.



Rysunek 3.13 Rozkład różnic wyników zawodników przy ocenie kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

Statystycznie we wszystkich scenariuszach wyniki są niemal identyczne. Współczynniki korelacji nie spadły poniżej 0.998. W przypadku nieznacznej zmiany kompilatora żaden wynik nie zmienił się o więcej niż 25 punktów. Przy zmianie komputera oceniającego, wynik jednego zawodnika zmienił się o 53 punkty, z 243 na 296, kolejnego o 50 punktów, z 320 na 270, a jeszcze innego o 48, z 212 na 260. Pozostałe wyniki nie zmieniły się więcej niż o 35 punktów. W przypadku zastosowania modelu procesora wyniki 10-ciu zawodników zmieniły się o więcej niż 35 punktów. W kolejnych punktach, poświęconych poszczególnym zadaniom, będziemy mogli zobaczyć, skąd wzięły się te zmiany.

3.3.3. Opis wykresów dla poszczególnych zadań

W dalszej części rozdziału przedstawiam trochę bardziej szczegółową analizę rozwiązań dla poszczególnych zadań. Tutaj chciałbym opisać rodzaje wykresów, jakie są użyte w kolejnych punktach.

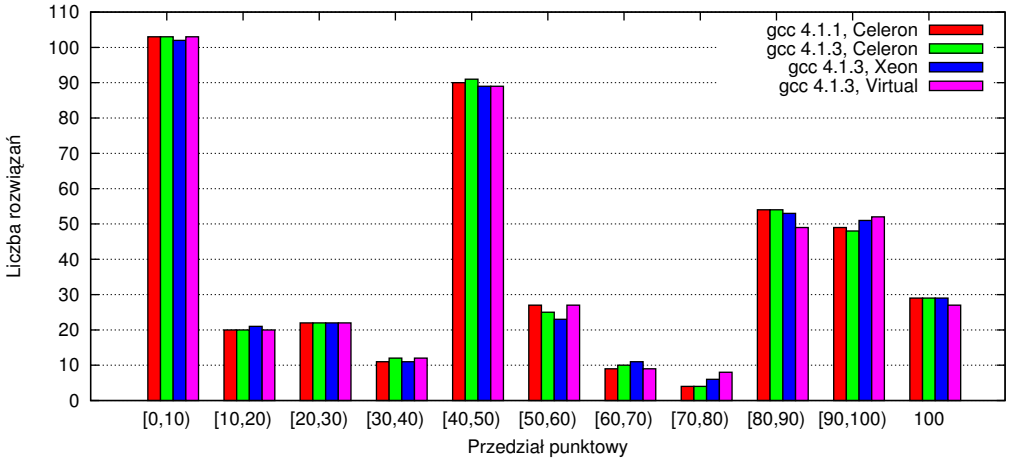
1. Histogram przedstawiający rozkład punktów uzyskanych w danym zadaniu.
2. Wykres, który dla poszczególnych programów porównuje trzy zmienne (patrz przykładowo rys. 3.15):
   - wynik punktowy przy ocenie kompilatorem gcc 4.1.3 na komputerze Celeron (czyli bieżącej maszynie sprawdzającej OI),
   - wynik punktowy przy ocenie kompilatorem gcc 4.1.3 na wirtualnym komputerze,
   - średnia liczba instrukcji, jakie program wykonał w ciągu sekundy na poszczególnych testach na komputerze Celeron.

   Średnia liczba instrukcji znajduje się na osi Y. Na osi X jest pierwsza z wymienionych zmiennych, a wielkość punktu jest proporcjonalna do różnicy wyników we wspomnianych środowiskach. Kolor czerwony oznacza, że program otrzymał mniej punktów na komputerze wirtualnym, a zielony, że więcej. Kolorem niebieskim oznaczono rozwiązania, których wynik się nie zmienił. Krzyżykiem zaznaczono rozwiązania przygotowane przez opracowującego i weryfikującego dane zadanie, a więc są to rozwiązania wzorcowe, nieoptymalne oraz niepoprawne.
3. Wykresy porównujące punktację poszczególnych programów. Są to wykresy podobne do rysunków 3.9–3.13. Obok nich znajdują się histogramy przedstawiające rozkład zmian wyników. Uwzględniono na nich jedynie rozwiązania, których wynik się zmienił.

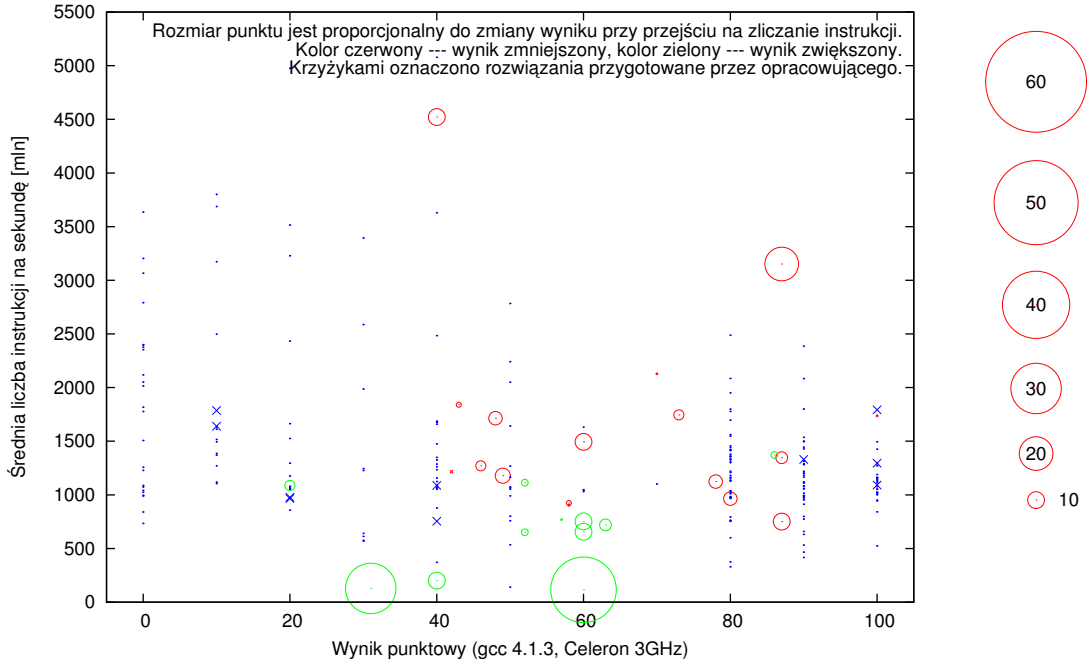
   Na osiach znajdują się wyniki punktowe przy ocenie w porównywanych środowiskach. Każdemu programowi odpowiada jeden punkt wykresu z tym, że kilka punktów w tym samym miejscu zostało zastąpionych proporcjonalnie większym symbolem.

3.3.4. Porównanie wyników dla zadania „Gaśnice”

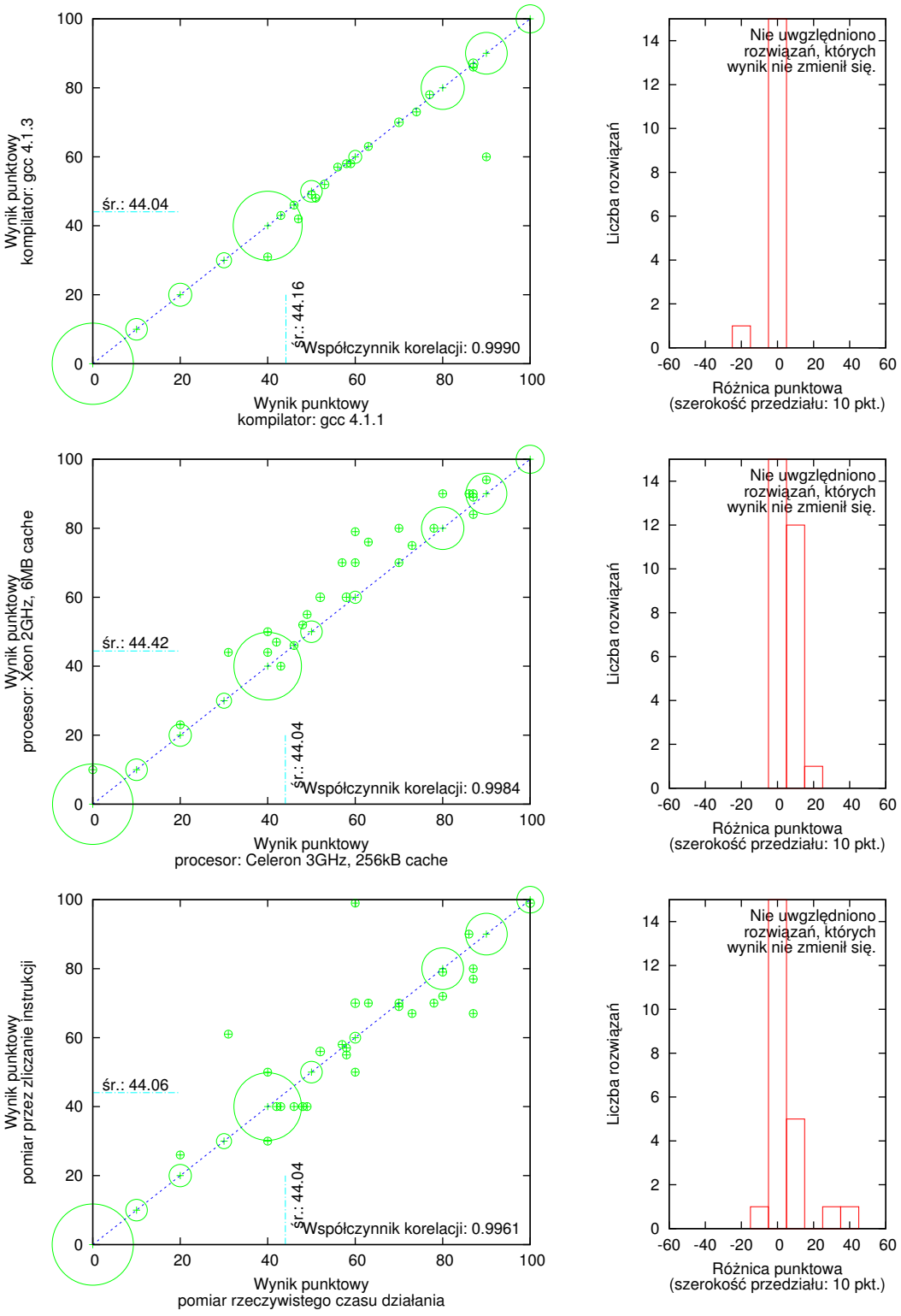
W tym zadaniu różnice punktowe pomiędzy różnymi metodami nie są duże. Jedno rozwiązanie dostaje 40 punktów więcej przy ocenie na modelu procesora. Tą sytuację można by poprawić, gdyby ustawić ostrzejsze limity czasowe. Dotychczas były one bardzo luźne ze względu na rozwiązania w Javie, wielokrotnie wolniejsze od takich samych rozwiązań w C++ lub Pascalu.



Rysunek 3.14 Rozkład punktów dla zadania Gaśnice.



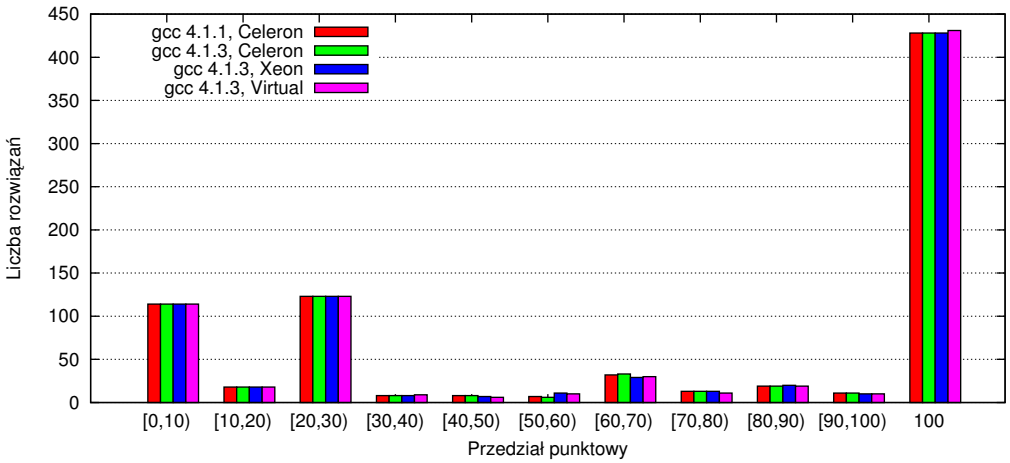
Rysunek 3.15 Porównanie wyników oraz szybkości względnej programów dla zadania Gaśnice.



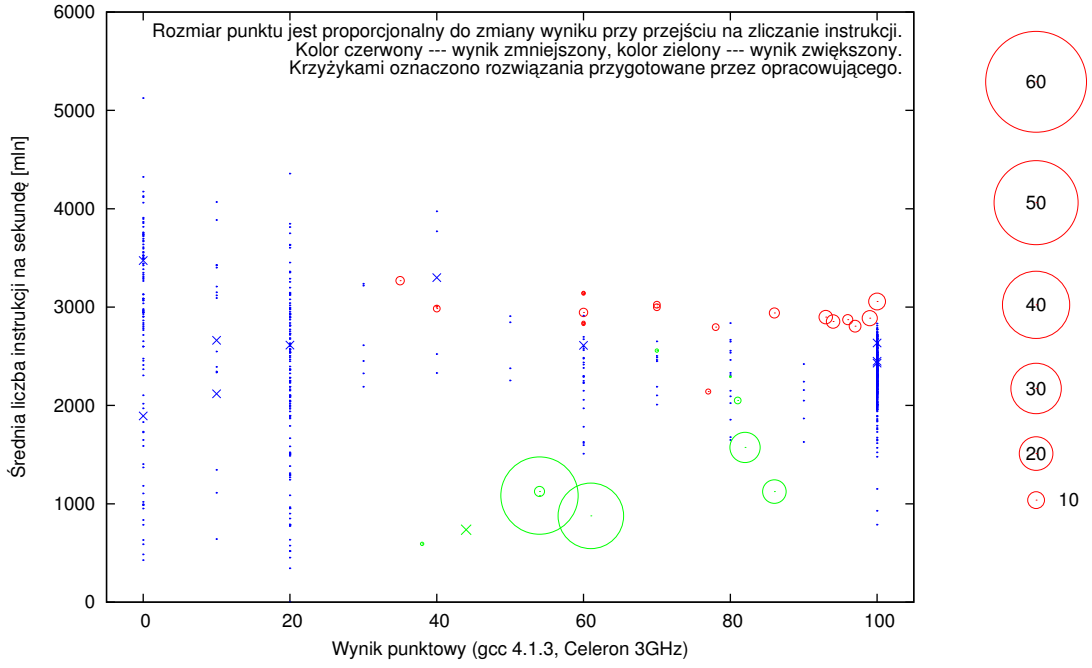
Rysunek 3.16 Porównanie wyników zadania Gaśnice przy ocenie (a) kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron, (b) kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon, (c) kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

3.3.5. Porównanie wyników dla zadania „Słonie”

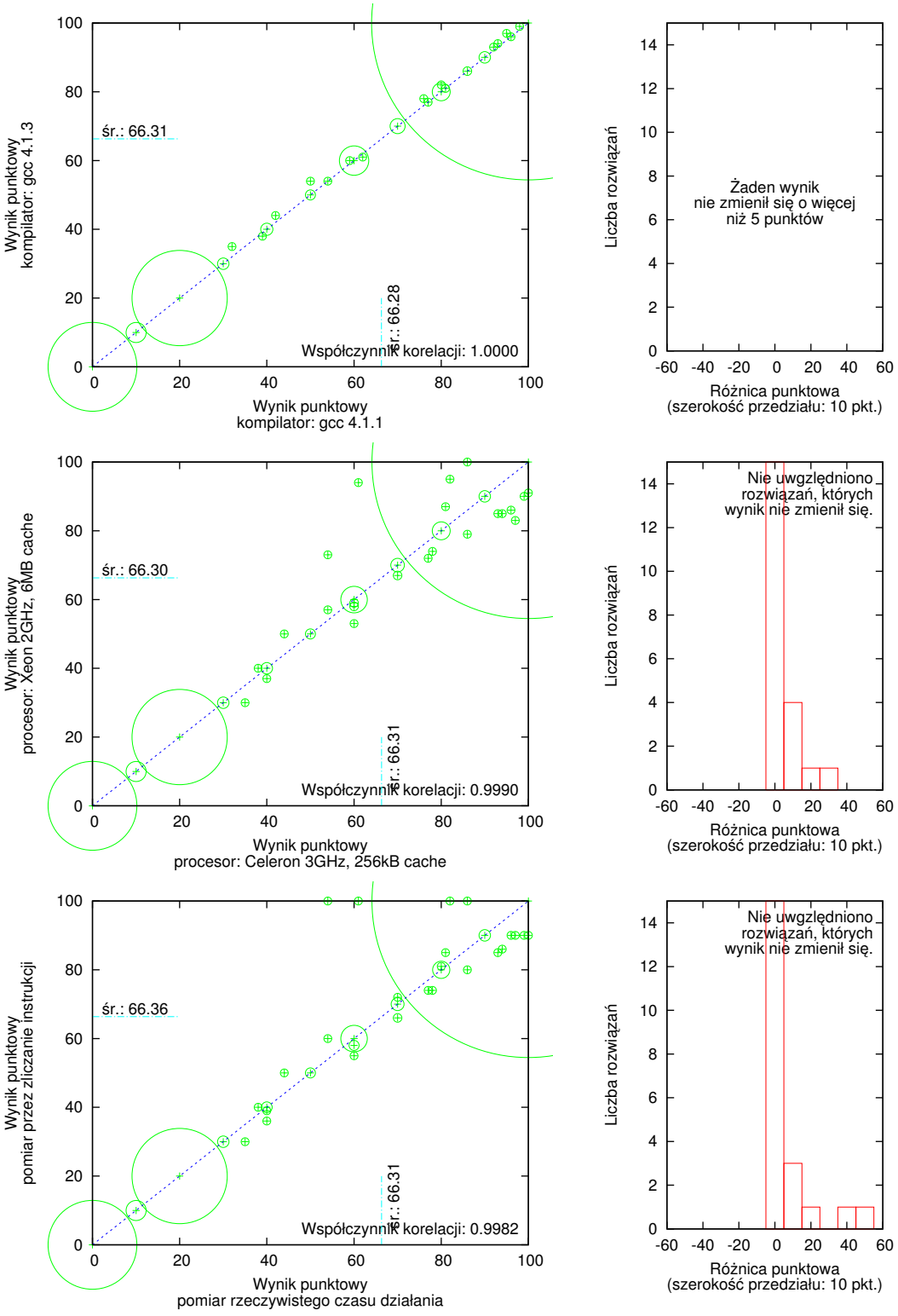
Z ciekawych zjawisk możemy zobaczyć dwa programy, których wynik znacznie się polepszył zarówno na komputerze Xeon, jak i na wirtualnym. Są to rozwiązania o złożoności większej o czynnik logarytmiczny od oczekiwanej i przez to trudno rozróżnialne od rozwiązań wzorcowych.



Rysunek 3.17 Rozkład punktów dla zadania Słonie.



Rysunek 3.18 Porównanie wyników oraz szybkości względnej programów dla zadania Słonie.



Rysunek 3.19 Porównanie wyników zadania Słonie przy ocenie (a) kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron, (b) kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon, (c) kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

3.3.6. Porównanie wyników dla zadania „Straż pożarna”

To jest zadanie, w którym najwięcej programów zmieniało swoje wyniki przy zmianie środowiska oceny. Już histogram (rys. 3.20) pokazuje, że sama zmiana wersji kompilatora spowodowała wzrost punktacji prawie stu programów o 10 punktów.

Dużo bardziej „spektakularne” efekty możemy zaobserwować na rys. 3.22:

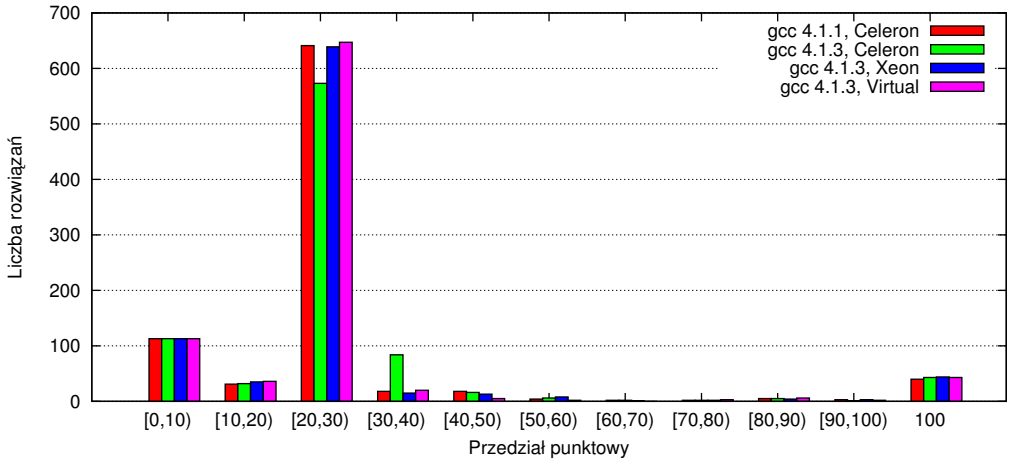
1. Kilka rozwiązań, które na komputerze Celeron dostawały około 40 punktów, na komputerze Xeon dostają po 90 – 100 punktów, a na komputerze z wirtualnym procesorem 80 – 90 punktów.

   Są to rozwiązania o nieoptymalnej złożoności, ale osiąganej dla bardzo specyficznych warunków, rzadko występujących w danych testowych. Wykorzystują drzewo czwórkowe do ograniczenia przestrzeni przeszukiwania, jednak w taki sposób, który nie gwarantuje szybkiego działania.

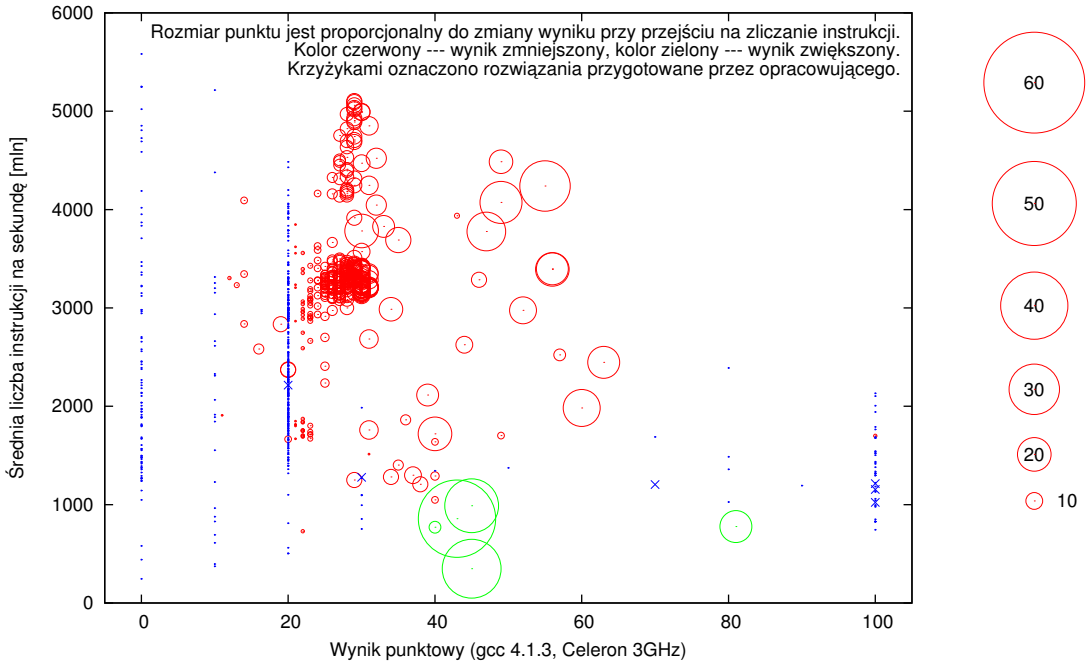
2. Grupa rozwiązań, które na komputerze Celeron dostawały 40 – 70 punktów, na komputerze z wirtualnym procesorem dostają 30 – 40 punktów mniej, a na komputerze Xeon 10 – 20 punktów mniej.

   Po bliższym przeanalizowaniu kodów tych rozwiązań wnioskuję, że są to rozwiązania hybrydowe, działające dla kilku przypadków tak, jak rozwiązanie wzorcowe, a dla pozostałych w dużo większej złożoności. Fakt, iż część nieefektywna jest prostą pętlą wykonującą szybkie instrukcje sprawia, że takie rozwiązanie dostaje więcej punktów, gdy jako podstawę oceny bierzemy czas rzeczywisty.

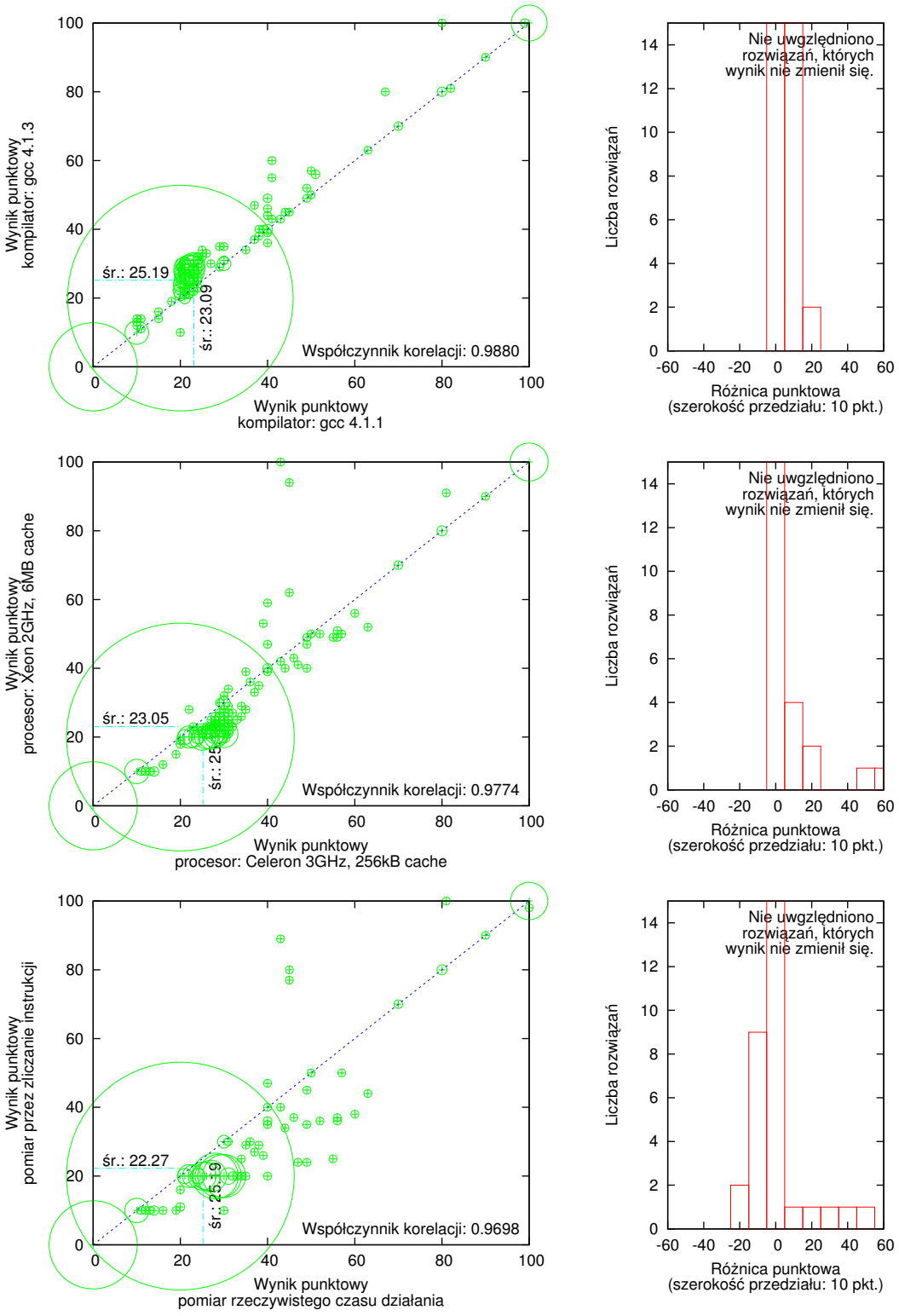
Ten przykład raz jeszcze potwierdza hipotezę, iż model procesora jest zbliżony do rzeczywistego procesora z olbrzymią pamięcią podręczną.



Rysunek 3.20 Rozkład punktów dla zadania Straż pożarna.



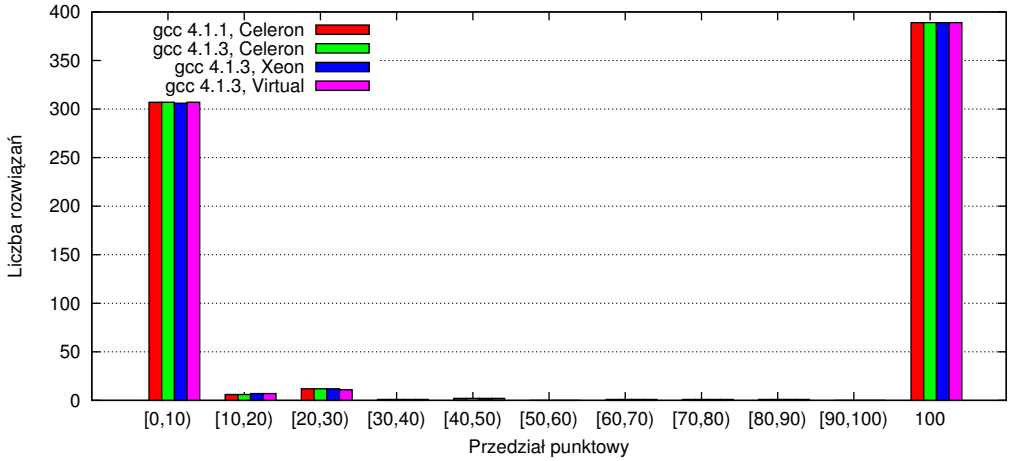
Rysunek 3.21 Porównanie wyników oraz szybkości względnej programów dla zadania Straż pożarna.



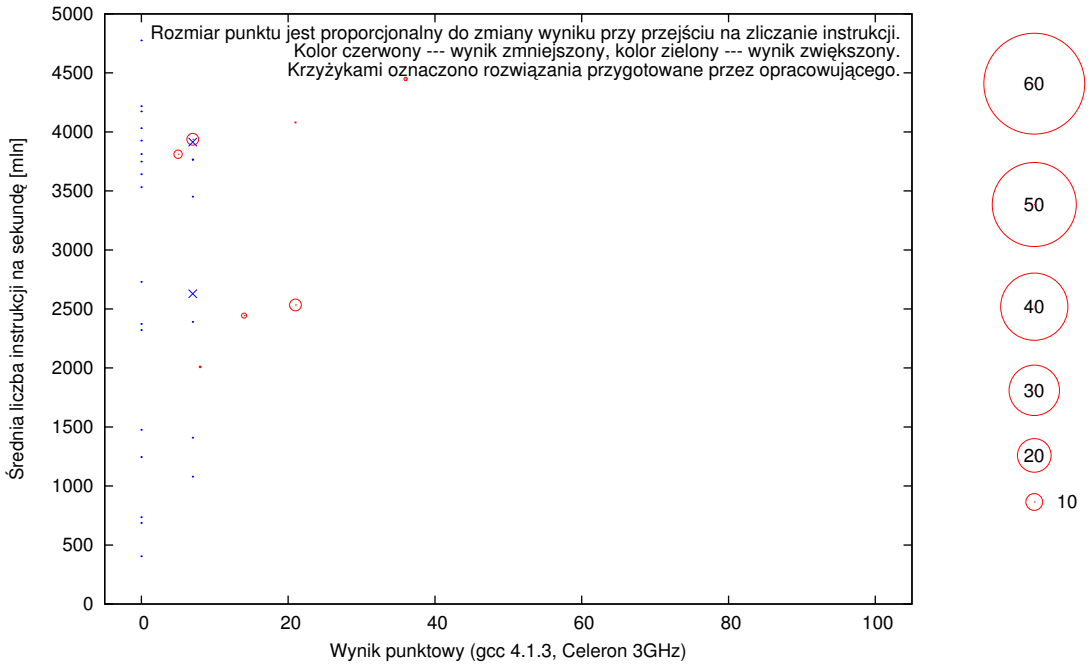
Rysunek 3.22 Porównanie wyników zadania Straż pożarna przy ocenie (a) kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron, (b) kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon, (c) kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

3.3.7. Porównanie wyników dla zadania „Kamyki”

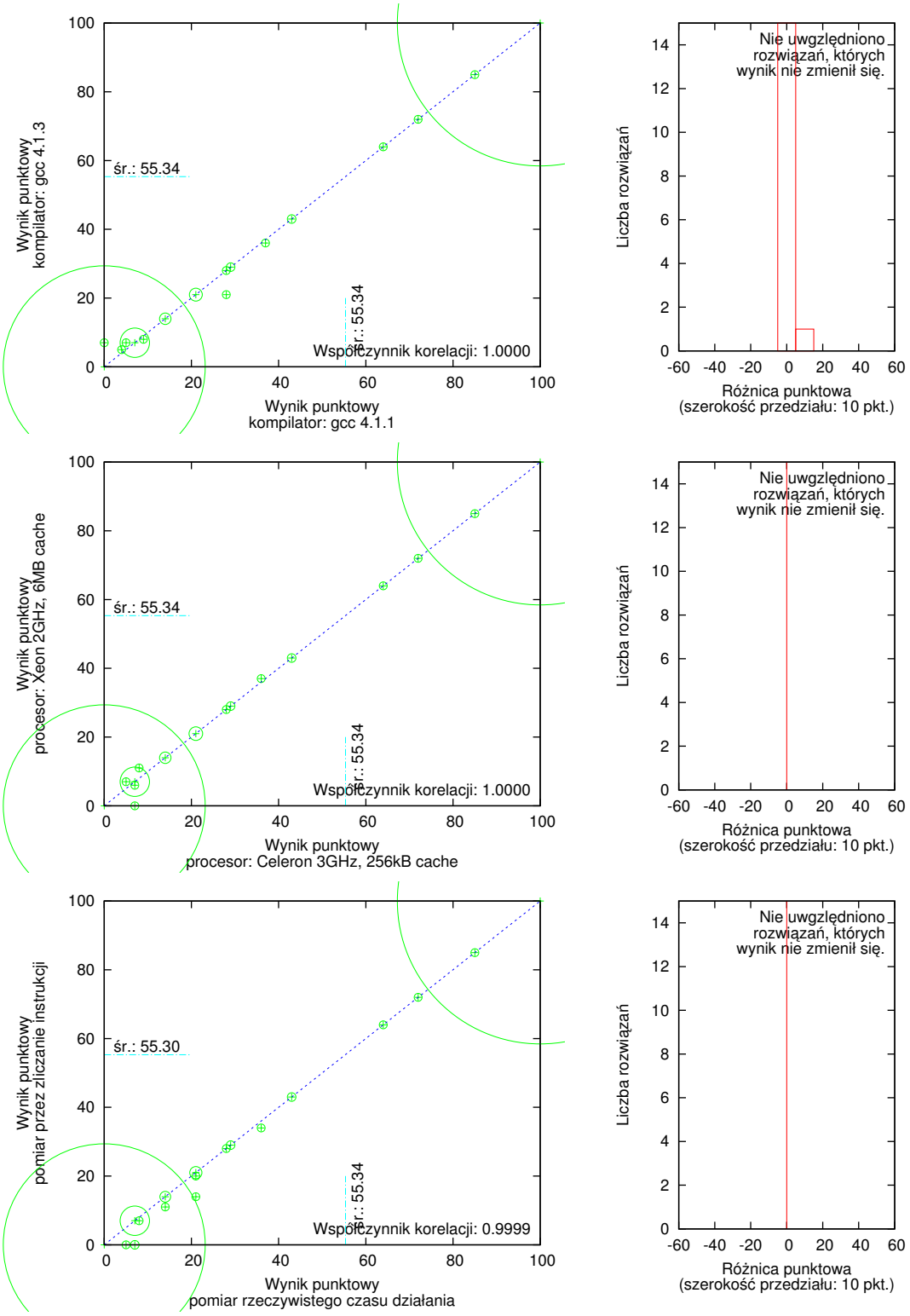
Okazuje się, że dla tego zadania wyniki praktycznie nie zależą od sposobu oceny. Było to proste zadanie, w którym zdecydowana większość rozwiązań była albo w pełni poprawna, albo dostawała 0 punktów. Pośrednią punktację otrzymywały rozwiązania wykładnicze.



Rysunek 3.23 Rozkład punktów dla zadania Kamyki.



Rysunek 3.24 Porównanie wyników oraz szybkości względnej programów dla zadania Kamyki.



Rysunek 3.25 Porównanie wyników zadania Kamyki przy ocenie (a) kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron, (b) kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon, (c) kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

3.3.8. Porównanie wyników dla zadania „Przyspieszenie algorytmu”

To zadanie jest o tyle ciekawe, że posiada rekordowe limity czasowe na największych testach — 240 sekund. Jest to trudne zadanie, dla którego otrzymaliśmy niewiele rozwiązań o oczekiwanej efektywności.

Wyniki eksperymentów pokazują dwa zjawiska:

1. Histogram wskazuje, że zmiana maszyny sprawdzającej, nawet bez użycia modelu procesora, spowodowała spadek punktacji ponad 40 rozwiązań z około 20 na około 10 punktów mimo ustawienia limitów czasowych zgodnie z wytycznymi w treści zadania. Te rozwiązania nie zmieściły się w jednosekundowym limicie czasowym dla jednego z małych testów po zmianie procesora na wolniejszy.

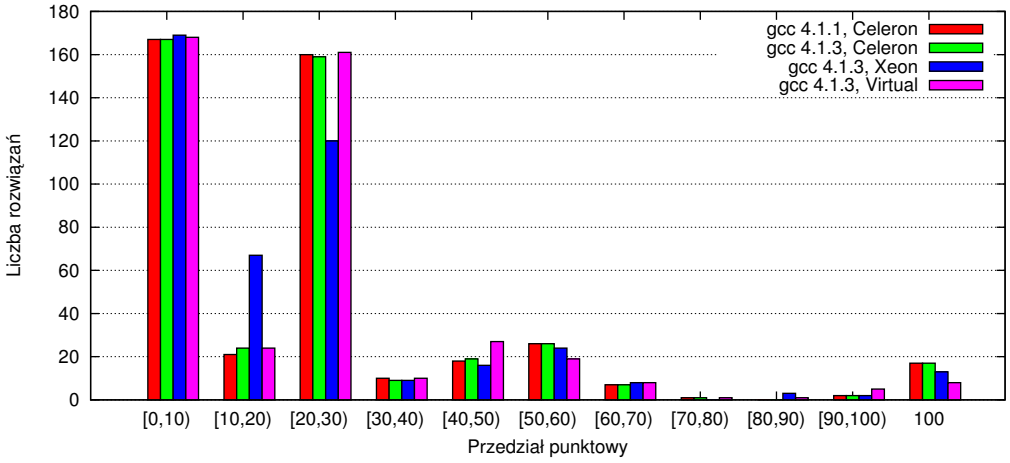
2. 6 rozwiązań, których punktacja oryginalnie była bliska 100% po zmianie procesora na wirtualny straciło od 30 do 60 punktów. Po bliższym przyjrzeniu się tej sytuacji okazało się, że:

   - wszystkie te rozwiązania posiadały zdecydowanie gorszą złożoność czasową niż oczekiwana, czyli zgodnie z intencją opracowującego nie powinny one dostać dużo punktów,
   - implementacja przygotowanego rozwiązania wzorcowego, choć wydaje się bardzo prosta, była wyjątkowo nieefektywna. Widać to na rys. 3.27. Rozwiązanie wzorcowe oznaczone krzyżykiem w prawym dolnym rogu wykonuje jedynie 213 mln. instrukcji w ciągu sekundy. Jedynie trzy inne programy w tym zadaniu wykonują mniej. Okazuje się, że winnym tak niskiej wartości są liczne operacje na tablicach, które prawie kompletnie uniemożliwiają efektywne korzystanie z pamięci podręcznej procesora. Tak więc tutaj widzimy przykład programu, i to wzorcowego, na którego czas działania duży wpływ ma pamięć podręczna procesora.

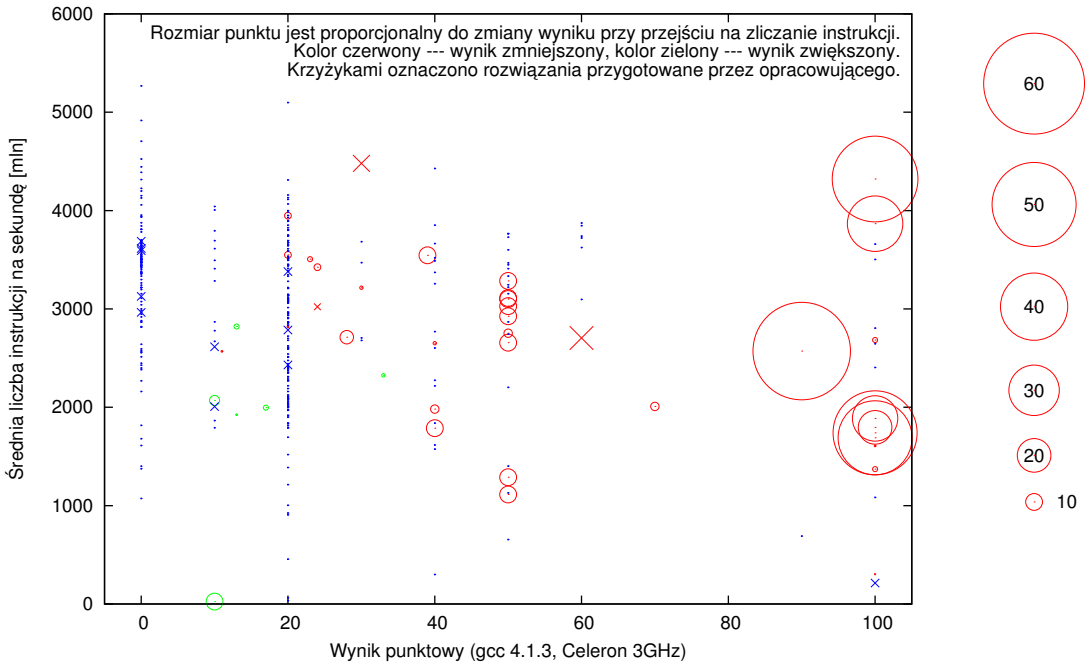
     Można by przypuszczać, że w takim razie zmiana komputera użytego go sprawdzania na taki, który ma więcej pamięci podręcznej, powinna mieć podobny efekt jak użycie modelu procesora, który nie uwzględnia zjawisk związanych z cachem. Istotnie, na rys. 3.28b widzimy z prawej strony programy, których wynik podobnie zmniejszył się na komputerze Xeon, który ma 6MB cache'u (Celeron ma 128kB).

Powyższa obserwacja jest godna podkreślenia, dlatego powtórzę ją raz jeszcze w formie dobrze widocznej ramki:

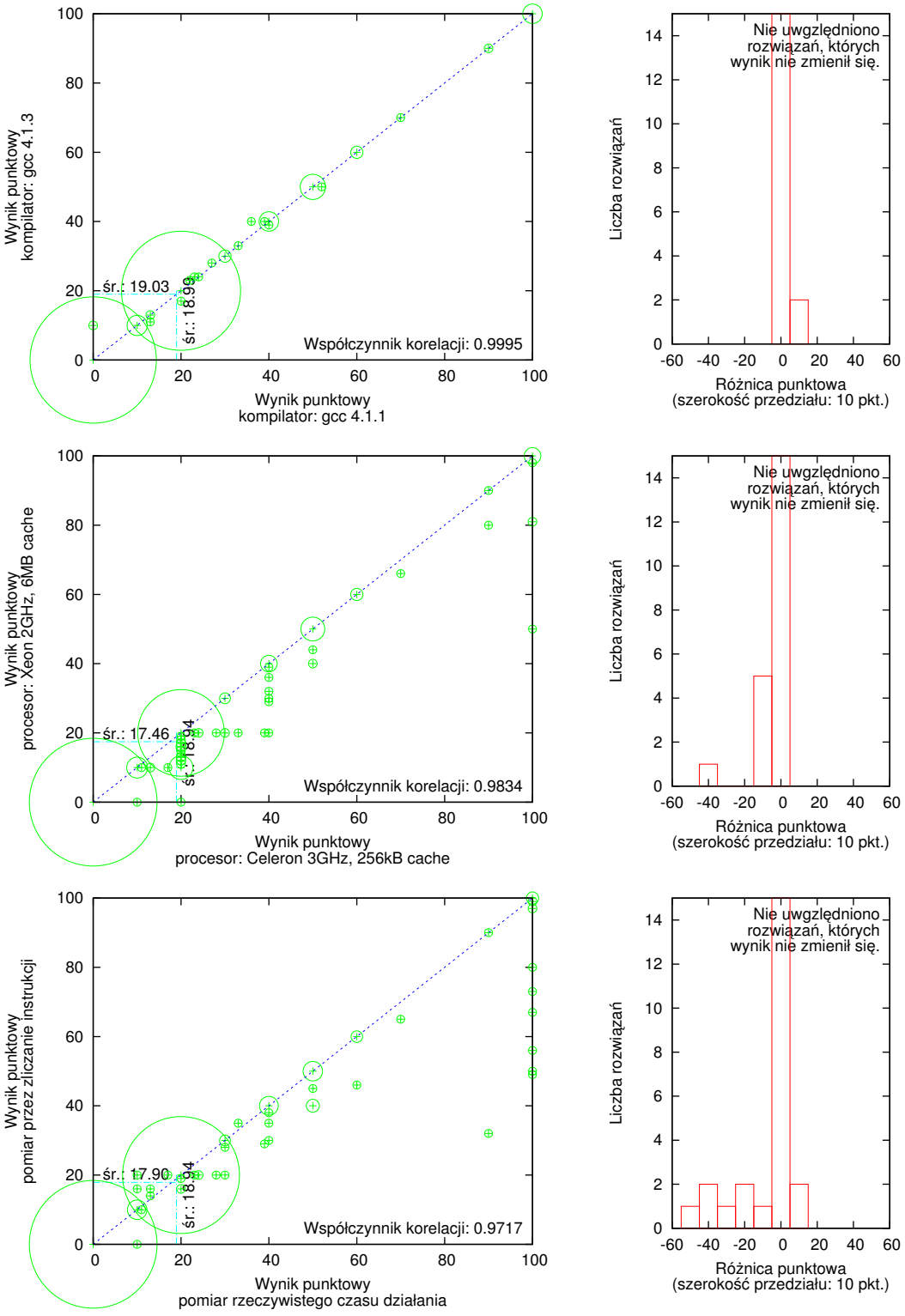
> Wyjątkowo nieefektywna implementacja rozwiązania wzorcowego może skutkować akceptacją rozwiązań o gorszych złożonościach. Wykorzystanie modelu procesora w dużym stopniu niweluje to zjawisko w sytuacjach, gdzie powodem nieefektywności implementacji była jej zła interakcja z pamięcią podręczną procesora.



Rysunek 3.26 Rozkład punktów dla zadania Przyspieszenie algorytmu.



Rysunek 3.27 Porównanie wyników oraz szybkości względnej programów dla zadania Przyspieszenie algorytmu.



Rysunek 3.28 Porównanie wyników zadania Przyspieszenie algorytmu przy ocenie (a) kompilatorem gcc 4.1.1 oraz 4.1.3 na komputerze Celeron, (b) kompilatorem gcc 4.1.3 na komputerze Celeron oraz Xeon, (c) kompilatorem gcc 4.1.3 na komputerze Celeron oraz na komputerze z wirtualnym procesorem.

The way to get started
is to quit talking and begin doing.

Walt Disney

Rozdział 4

Wnioski

Zmiany wersji kompilatora czy konfiguracji maszyn sprawdzających były do tej pory normalnym zjawiskiem, nie wymagającym konsultacji z Komitetem Głównym. O konfiguracji komputerów sprawdzających zawodnicy nie są informowani. Przedstawione eksperymenty wskazują podobieństwo wprowadzenia modelu procesora do zmiany modelu fizycznego procesora użytego do oceny, choć różnice punktowe w tym pierwszym przypadku są trochę większe. Tym niemniej nie zaobserwowano zjawisk, które wzbudzałyby zaniepokojenie co do jakości oceny rozwiązań. Nie można jednak wyciągnąć ostatecznych wniosków na podstawie danych pochodzących z analizy wyników dla jedynie pięciu zadań.

W tym miejscu chciałbym więc przytoczyć krótkie porównanie „za" i „przeciw" dla proponowanej metody. Bardziej szczegółowy opis problemów, które rozwiązuje modelowanie procesora, znajduje się w punkcie 2.1 (str. 13).

ZA

- eliminacja konieczności identycznej konfiguracji maszyn sprawdzających oraz konieczności ponownego ustawiania limitów czasowych w przypadku wymiany tych maszyn,
- niezależność wyników od konfiguracji sprzętowej komputerów sprawdzających,
- powtarzalność oceny zadań, np. w serwisie `www.main.edu.pl`,
- możliwość ustalenia ostatecznych limitów czasowych na etapie opracowania zadania,
- eliminacja pamięci podręcznej procesora, a przez to zmniejszenie wpływu na wyniki niekontrolowanego elementu zależnego od konkretnego procesora.

PRZECIW

- większe różnice wyników przy przejściu na wirtualny procesor niż przy przejściu na inny fizyczny procesor,
- koszty związane z przeprowadzeniem dalszych badań oraz przygotowaniem oprogramowania i szkoleniem jurorów,
- ryzyko negatywnego odzewu ze strony środowiska.

- możliwość współdzielenia maszyn sprawdzających z innymi usługami,
- możliwość uruchomienia oprogramowania sprawdzającego na dowolnych komputerach, także na wirtualizowanych,

Jak widzimy, modelowanie procesora ma duży potencjał do ulepszenia procesu przygotowania zadania, jego oceny i potem udostępnienia zawodnikom do treningu. Gdyby jeszcze udało się dostarczyć zawodnikom programy umożliwiające uruchamianie własnych programów na własnych komputerach, i mierzenie czasu ich działania w modelu, byłoby to pierwsze rozwiązanie umożliwiające zawodnikom dokładną duplikację środowiska oceny na własnych stanowiskach. Niewątpliwie dodałoby to wartości Olimpiadzie.

Dlatego uważam, że dalsza praca nad modelowanie procesora oraz dążenie do wprowadzenia go jako metody oceniania zadań na Olimpiadzie Informatycznej jest warta uwagi i dalszej pracy. W kolejnych punktach prezentuję przykładowy plan przygotowania zmian oraz serię uwag, o których warto pamiętać.

4.1. Propozycja planu zmian

Proponuję następujący plan dalszej pracy. Liczę, że pozwoli on dobrze przygotować się do zmian, wziąć pod uwagę opinie wszystkich zainteresowanych stron i świadomie podjąć decyzję o wprowadzeniu bądź niewprowadzeniu zmian.

1. Rok szkolny 2009/10, etap I i II — powtórzyć eksperymenty zawarte w tej pracy, aby uzyskać dokładniejszy obraz wpływu nowej metody oceny na wyniki. Wykonać eksperymenty dla II i III etapu minionej olimpiady.
2. Rok szkolny 2009/10, etap III — wprowadzić podwójne ocenianie, na następujących zasadach:
   - na każdym teście program zawodnika jest oceniany zarówno tradycyjną, jak i nową metodą,
   - każdy test posiada osobne limity czasowe dla obydwu metod,
   - wynik programu na teście jest lepszym z dwóch obliczonych wyników.

   W ten sposób można bez nadmiernego zaburzania oceniania nabrać doświadczenia w wykorzystaniu modelu do oceny na zawodach oraz zebrać opinie od zawodników.

   Zaproponowałem, by zrobić to po raz pierwszy na finale z dwóch powodów: po pierwsze zmiana dotyczyłaby niewielkiej liczby zawodników, a po drugie na finale przeważają zawodnicy doświadczeni, którzy lepiej wychwycą różnice w sposobach oceniania, niż ci, który dopiero zaczynają startować w olimpiadzie.
3. Wakacje 2010 — opracowanie danych, dopracowanie modelu, jeśli zajdzie taka konieczność, stworzenie narzędzi dla jurorów, szkolenie jurorów (opracowujących zadania).
4. Rok szkolny 2010/11, etap I i II — w dalszym ciągu podwójne ocenianie, zbieranie opinii od uczestników. Dyskusja w gronie Komitetu, podjęcie decyzji co do dalszych losów projektu. W przypadku ogólnego poparcia, rezygnacja z pomiaru czasu rzeczywistego począwszy od III etapu.

4.2. Uwagi dotyczące sposobu wprowadzenia zmian

Niniejszy punkt chciałbym przeznaczyć na przekazanie kilku uwag dotyczących przygotowania zmian i przekazania informacji o niej zawodnikom. Warto pomyśleć, jak zapewnić, żeby nowy system nie był postrzegany jako zmiana tylko i wyłącznie ułatwiająca obsłudze przeprowadzenie zawodów, bądź zmniejszająca koszty organizatorów, i to na dodatek uderzająca w zawodników.

Pozwoliłem dobie wypisać poniżej zagadnienia, których lepiej nie pominąć przy wprowadzaniu zmian:

1. Zapewnienie, żeby zawodnicy mieli łatwy dostęp do narzędzi mierzących czas według modelu przed zgłoszeniem rozwiązania do oceny. Do tej pory zawodnicy mieli w SIO możliwość uruchamiania swoich programów na własnych danych testowych w środowisku ostatecznej oceny. Poza tym mogli po prostu uruchomić program na swoim własnym komputerze, zmierzyć czas jego działania i założyć, że na komputerze oceniającym wynik będzie zbliżony z dokładnością do stałej.

   Obie metody w prosty sposób działają również po zmianie metody liczenia czasu, o ile nie okaże się, że zależność między rzeczywistym czasem działania programów a wynikiem w systemie sprawdzającym nie odbiega zbyt bardzo od liniowej.

   Niezależnie od tego, najlepszym wyjściem byłoby umożliwienie zawodnikom uruchamiania własnych programów na własnych komputerach **oraz** pomiar czasu według modelu. Żaden z przedstawionych wcześniej produktów, za wyjątkiem biblioteki Pin, nie pozwala na jednolity pomiar liczby instrukcji zarówno pod Windowsem, jak i Linuxem, jednocześnie zachowując łatwość instalacji. Wadą Pina jest to, że jest dostępny jedynie dla procesorów Intel.

2. Przeprowadzenie zmiany od strony informacyjnej powinno być spokojne, aby nie wzbudzać niepotrzebnych emocji, lecz zachęcić do obiektywnej oceny. Na przykład główny przekaz może mieć postać dodatkowego punktu w dokumencie „Ustalenia techniczne" i przykładowo brzmieć tak:

   > Programy zawodników uruchamiane są w (wirtualnym) środowisku, w którym procesor wykonuje dokładnie 4 mld. instrukcji w ciągu sekundy.

   a nie tak:

   > Ocenie podlega liczba instrukcji wykonanych przez program zawodnika.

   Choć obie przedstawione wypowiedzi przekazują tą samą informację, ta pierwsza wydaje się mniej prowokować do nieprzemyślanych protestów.

   Warto też w widocznym miejscu witryny OI umieścić krótką notatkę o zmianie ze wskazaniem miejsca, z którego można pobrać narzędzia do samodzielnego uruchamiania programów pod modelowanym procesorem (jeśli takie powstaną) oraz zwrócić uwagę na możliwość testowania własnych programów na własnych danych testowych przez SIO.

3. Stopniowe wprowadzenie tej samej metody oceny w serwisach z zadaniami. W końcu jedną z zalet nowego sposobu oceny jest to, że niezależnie od serwisu, na którym zadanie jest oceniane, wynik powinien być taki sam. Aczkolwiek należy uważać na to, że będzie to prawdą tylko wtedy, gdy zawsze będzie używany ten sam kompilator.

Podsumowanie

Na początku pracy przedstawiono środowisko konkursów algorytmicznych w Polsce i na świecie i przedstawiono sposób przygotowania zadania na zawody Olimpiady Informatycznej. Ten wstęp miał na celu wprowadzić czytelnika w kontekst pracy i przedstawić sposób oceny zadań na Olimpiadzie Informatycznej i podobnych zawodach.

W kolejnym rozdziale omówiono współczesne techniki modelowania procesora oraz przedstawiono prosty model przeznaczony do wykorzystania w ocenianiu zadań na zawodach algorytmicznych. Wykorzystanie modelu procesora pozwala uniezależnić wyniki oceny od konfiguracji sprzętowej komputera użytego do przeprowadzania oceny. Także nie wymaga jego specjalnej konfiguracji oraz umożliwia współdzielenie z innymi usługami. Wprowadza kompatybilność z technologiami wirtualizacji — nie wymaga fizycznego komputera na wyłączność, jak dotychczasowa metoda pomiaru czasu. Te cechy pozwalają na wprowadzenie nowatorskich rozwiązań służących zawodnikom i organizatorom (patrz rozdział 4).

Następnie opisano wyniki eksperymentów dwojakiego rodzaju:

- przedstawiających różnice w szybkości działania takich samych programów na komputerach nieznacznie różniących się procesorem. W warunkach laboratoryjnych stworzono program, którego czas działania potrafił się różnić sześćdziesięciokrotnie na dwóch procesorach o podobnej częstotliwości zegara. W warunkach rzeczywistych znaleziono rozwiązanie wzorcowe jednego z zadań olimpijskich, które różniło się czasem działania na tych samych dwóch procesorach siedmiokrotnie.

- polegających na powtórzeniu oceny I etapu XVI Olimpiady Informatycznej przy użyciu zaproponowanego modelu, jak również przy użyciu dwóch rzeczywistych procesorów. Wyniki potwierdziły hipotezę, iż model procesora zachowuje się bardzo podobnie do rzeczywistego procesora z bardzo dużą ilością pamięci podręcznej. Nie zaobserwowano niepokojących zjawisk.

Ze względu na optymistyczne wyniki, w rozdziale 4 zaproponowano stopniowe wprowadzenie modelowania procesora do oceny zadań na Olimpiadzie Informatycznej.

Podziękowania

Chciałbym serdecznie podziękować przede wszystkim mojemu promotorowi, a jednocześnie przewodniczącemu Komitetu Głównego Olimpiady Informatycznej, prof. Krzysztofowi Diksowi za nieocenioną pomoc w realizacji badań i tworzeniu niniejszej pracy.

Dziękuję mojemu bratu Kubie, jak również Konradowi Gołuchowskiemu i Przemkowi Horbanowi za cenne dyskusje i pomoc edytorską. Dziękuję moim rodzicom za nieustanną mobilizację i doping!

Dodatek A

Przykład opracowania zadania na Olimpiadę Informatyczną

Na kolejnych stronach umieszczono przykładowe dokumenty, które powstają podczas procesu opracowania zadania na Olimpiadę Informatyczną. Jako przykład wybrano zadanie *Gaśnice* z I etapu ostatniej, XVI OI. Więcej informacji dotyczących procedury przygotowania zadania można znaleźć w punkcie 1.2 niniejszej pracy.

W procesie przygotowania zadania zazwyczaj powstają następujące dokumenty:

- treść zadania — początkowo przygotowana w fazie *redakcji*, dopracowywana podczas *opracowania* oraz *weryfikacji*,
- dokument opracowania — zawierający opis rozwiązania wzorcowego, rozwiązań alternatywnych, nieefektywnych oraz niepoprawnych. Zawiera również wskazówki dotyczące doboru limitów czasowych i punktacji poszczególnych testów. Powstaje w fazie *opracowania*. Bywa kosmetycznie zmieniany podczas *weryfikacji*.
- dokument weryfikacji — krótki raport zawierający informacje o czynnościach wykonanych w fazie *weryfikacji*.

W specyficznych sytuacjach mogą powstać również inne:

- tłumaczenia treści zadań — w przypadku zmiany przeznaczenia zadania na zawody międzynarodowe lub w przypadku publikacji w języku obcym,
- recenzje — w przypadku organizacji zawodów międzynarodowych, w którym kraje uczestniczące w zawodach mogą proponować zadania. Tak się dzieje między innymi na Międzynarodowej Olimpiadzie Informatycznej.
- opisy rozwiązań — czasem opis rozwiązania zawarty w dokumencie opracowania jest przepisywany w bardziej przystępnej formie, w celu opublikowania w „Niebieskiej książeczce” [10].

Zadanie: GAS
Gaśnice

XVI OI, etap I. Dostępna pamięć: 64 MB. *20.10–17.11.2008*

Bajtazar zbudował nowy pałac. Pałac ten składa się z n pokoi i $n-1$ łączących je korytarzy. Pokoje są ponumerowane od 1 do n. Do pałacu jest jedno wejście, prowadzące do pokoju nr 1. Każdy korytarz łączy dwa różne pokoje. Do każdego pokoju prowadzi od wejścia dokładnie jedna droga (bez zawracania). Inaczej mówiąc, pokoje i łączące je korytarze tworzą *drzewo* — spójny graf acykliczny.

Inspektor straży pożarnej, który odbierał pałac, domaga się umieszczenia w pałacu gaśnic. Określił on następujące wymagania:

- Gaśnice mają być umieszczone w niektórych pokojach, przy czym w jednym pokoju może znaleźć się kilka gaśnic.
- Każdemu pokojowi trzeba przydzielić jedną konkretną gaśnicę.
- Każda z gaśnic może być przydzielona do gaszenia co najwyżej s różnych pokoi.
- Dotarcie z dowolnego pokoju do przypisanej mu gaśnicy może wymagać przejścia co najwyżej k korytarzy.

Bajtazar, jak to zwykle bywa po zakończeniu budowy, ma bardzo mało pieniędzy. Zastanawia się więc, jaka minimalna liczba gaśnic wystarczy do spełnienia powyższych wymagań?

Wejście

W pierwszym wierszu standardowego wejścia znajdują się trzy liczby całkowite n, s i k pooddzielane pojedynczymi odstępami, $1 \leqslant n \leqslant 100\,000$, $1 \leqslant s \leqslant n$, $1 \leqslant k \leqslant 20$. Każdy z następnych $n-1$ wierszy zawiera po dwie liczby całkowite oddzielone pojedynczym odstępem. W wierszu $i+1$ znajdują się liczby $1 \leqslant x_i < y_i \leqslant n$ reprezentujące korytarz łączący pokoje nr x_i i y_i.

Wyjście

W pierwszym i jedynym wierszu standardowego wyjścia powinna zostać wypisana jedna liczba całkowita — minimalna liczba gaśnic, jakie należy zainstalować w pałacu.

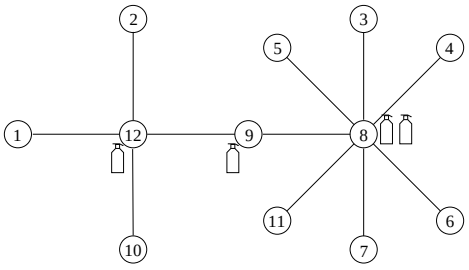
Przykład

Dla danych wejściowych:

```
12 3 1
1 12
3 8
7 8
8 9
2 12
10 12
9 12
4 8
5 8
8 11
6 8
```

poprawnym wynikiem jest:

```
4
```



Opracowanie: GAS
Gaśnice

HISTORIA:
- v. 1.01: 2008.09.29, BGOR, weryfikacja
- v. 1.00: 2008.09.14, PNIE, przygotowanie opracowania

dokument systemu SINOL 1.9.1

1 Rozwiązania optymalne-autorskie $O(n \cdot k)$

Najpierw wybieramy dowolny wierzchołek jako korzeń drzewa. Następnie drzewo będziemy przetwarzać od liści do korzenia, postępując „zachłannie": intuicyjnie w każdym poddrzewie stawiając jak najmniej gaśnic, a te które stawiamy chcemy postawić jak najwyżej, żeby obejmowały jeszcze jak najwięcej wierzchołków w górę. Dobrze jest wyobrażać sobie, że gaśnica ma s „końcówek", z których prowadzimy ścieżki długości $\leqslant k$ do pól.

Konkretnie rozmieszczenie gaśnic w danym poddrzewie będziemy charakteryzować poprzez następujący układ liczb: $(g, x_0, x_1, ..., x_k, y_0, y_1, ..., y_k)$, gdzie g to liczba gaśnic użytych w tym poddrzewie; y_i to liczba pól na głębokości i (korzeń jest na głębokości 0), które potrzebują przypisania gaśnicy; natomiast x_i to liczba wolnych końcówek gaśnic, które sięgają na i pól w górę ponad korzeń (dla przykładu jeśli ustawimy gaśnicę o 1 pod korzeniem i nic do niej nie przypiszemy, to x_{k-1} będzie równe s, bo gaśnica ta sięga jeszcze o $k-1$ pól ponad korzeń; jeśli już przypisane jest do niej jakieś j pól, to x_{k-1} będzie równe $s-j$). Zauważmy w szczególności, że (jeśli rozstawienie jest poprawne) $y_k = 0$, bo polom na głębokości k nie możemy przypisać gaśnicy spoza poddrzewa (zatem y_k jest w naszym ciągu tylko dla wygody).

Gdy wykonujemy algorytm w jakimś wierzchołku, to rozstawienie dla wszystkich poddrzew powinno być już policzone (działamy od liści do korzenia). Algorytm w wierzchołku działa w następujący sposób:

1. Najpierw sumujemy charakteryzacje wszystkich poddrzew; robimy to po współrzędnych, tzn. nowe g = suma wszystkich g z poddrzew, nowe x_0 = suma wszystkich x_0 z poddrzew, itd.
2. Przesuwamy ciągi (ponieważ korzeń poddrzewa jest teraz o 1 wyżej niż poprzednio). Jako nowe $x_0, x_1, ..., x_{k-1}$ przyjmujemy $x_1, x_2, ..., x_k$, za $y_1, y_2, ..., y_k$ przyjmujemy $y_0, y_1, ..., y_{k-1}$. Jako x_k przyjmujemy 0 (w korzeniu nie stoi na razie żadna gaśnica, więc nie ma wolnych końcówek wystających o k w górę). Jako y_0 przyjmujemy 1 (jest jedno pole na głębokości 0, czyli korzeń, które potrzebuje przypisania). Zapominamy o starym x_0 (te wolne końcówki i tak nie sięgają w górę, nawet do nowego korzenia) oraz o starym y_k (które i tak jest 0).
3. W korzeniu aktualnie badanego poddrzewa stawiamy $c = \lceil y_k/S \rceil$ gaśnic (czyli tyle, ile koniecznie jest potrzebnych), tzn. robimy $g+ = c$ oraz $x_k = S * c$.
4. Dla każdego i parujemy nieprzypisane pola na głębokości i z wolnymi końcówkami wystającymi o i w górę. Tzn. dla $c = min(x_i, y_i)$ robimy tak: $x_i- = c, y_i- = c$.
5. (ważne, że to robimy po 4). Dla każdego $i < k$ parujemy nieprzypisane pola na głębokości i z wolnymi końcówkami wystającymi o $i+1$ w górę. Czyli dla $c = min(x_{i+1}, y_i)$ robimy tak: $x_{i+1}- = c, y_i- = c$.

W ten sposób dostajemy wynikowy ciąg dla naszego poddrzewa, który przekazujemy w górę. Zauważmy, że będzie $y_k = 0$, tak jak chcieliśmy, bo po kroku 3 jest $x_k >= y_k$.

W korzeniu całego drzewa zamiast kroków 3-5 trzeba oczywiście postąpić nieco inaczej:

3') Teraz przeglądamy kolejne $i \in \langle 0..k \rangle$ i dla każdego i przeglądamy po kolei wszystkie $j \in \langle i \ldots k \rangle$. Ważna jest tutaj kolejność. Dla każdej takiej pary robimy: $c = min(y_i, x_j)$ i teraz $y_i- = c, x_j- = c$ Autor rozwiązania proponował robić ten punkt troszeczkę inaczej, ale różnica jest pomijalna, a to postępowanie jest według mnie bardziej intuicyjne.

4') Pozostałych końcówek nie da się wykorzystać, stawiamy nowe gaśnice, $c = \lceil (y_0 + ... + y_K)/S \rceil$ gaśnic oraz przypisujemy je tym polom, które pozostały, tzn. $g+ = c, x_k+ = S * c - (y_0 + ... + y_k), y_0 = ... = y_k = 0$.

Implementując to rozwiązanie tak wprost, jak jest opisane, dostajemy złożoność $O(n * k)$. Po dłuższym zastanowieniu nie widzę żadnego rozwiązania w $O(n)$, choć nie wykluczam jego istnienia (być może można jakoś wyjątkowo sprytnie wykonywać powyższe operacje); jednak nawet gdyby takie rozwiązanie istniało to raczej jest zbyt trudne do wymyślenia i myślę, że wymaganie złożoności $O(n * k)$ daje właściwy poziom trudności zadania.

Pozostaje udowodnić, że postępując „zachłannie" jak opisałem powyżej dostajemy optymalne rozwiązanie. Jak to często bywa przy rozwiązaniach zachłannych, dowód jest nieco przydługi. Dowód jest indukcyjny, dla coraz to większych poddrzew p będziemy dowodzić równocześnie, że:

1. Każde poprawne rozmieszczenie gaśnic w drzewie zawiera w poddrzewie p przynajmniej tyle gaśnic, ile zwraca nasz algorytm.

2. Mając poprawne rozmieszczenie gaśnic w całym drzewie możemy przeorganizować gaśnice w poddrzewie p tak aby stały zgodnie z naszym algorytmem (jeśli w p jest więcej gaśnic niż mówi nasz algorytm, to zbyteczne gaśnice wynosimy o jeden wierzchołek ponad korzeń p), a pozostawić w reszcie drzewa i dostaniemy poprawne rozstawienie. Dotyczy to także przypisań pól do końcówek. Żeby jednak być precyzyjnym należy sobie wyobrażać przypisania pól końcówkom jako ścieżkę długości $\leqslant k$, jednak niekoniecznie ścieżkę prostą. Ścieżka taka idzie najpierw w górę, a potem w dół. Nasz algorytm pewne przypisania wykonuje w obrębie poddrzewa p. Ścieżki od pozostałych, nieprzypisanych pól i nieprzypisanych końcówek prowadzi w górę, ponad p. Gdzieś tam poza p mogą one zostać ze sobą połączone, ale takie przypisanie traktujemy już jako przypisanie poza p. Tutaj chcemy, że przypisania zrobione przez nasz algorytm wewnątrz p mają być właśnie tak zrobione, a jeśli jakieś pola lub końcówki wychodzą w naszym algorytmie poza p, to tu też mają wychodzić poza p.

Aby dowieść to dla p, najpierw stosujemy założenie indukcyjne dla poddrzew p i przestawiamy w nich gaśnice, aby stały zgodnie z naszym algorytmem.

Zauważmy, że w korzeniu p trzeba postawić co najmniej $\lceil y_k/s \rceil$ gaśnic (tyle ile stawia nasz algorytm), bo polom na głębokości K trzeba przypisać gaśnicę, która stoi dokładnie tutaj. To dowodzi już pierwszej własności.

Aby dowieść drugiej własności trzeba jedynie rozważyć przypisania, bo rozstawienie gaśnic jest już jednoznaczne. Powiedzmy, że mamy jakiś inny układ przypisań i chcemy uzasadnić, że bez problemu możemy zmienić część z nich, aby uzyskać takie przypisania, jakich chce nasz algorytm. Interesują nas tylko przypisania, które przechodzą przez korzeń p, pozostałe są już załatwione z założenia indukcyjnego. Najpierw zauważmy, że jeśli jest pewne połączenie pewnego pola na głębokości i z pewną końcówką na poziomie j, i nasz algorytm też wymaga połączenia pewnego pola na głębokości i z pewną końcówką na poziomie j, ale być może innych, to możemy połączenia tych pól i końcówek pozamieniać, żeby było zgodnie z naszymi oczekiwaniami. To samo dotyczy pól lub końcówek na ustalonym poziomie nie połączonych z niczym wewnątrz p. Innymi słowy nie liczy się co dokładnie z czym jest połączone, liczy się tylko liczba połączeń danego rodzaju (rodzaj jest charakteryzowany przez głębokość łączonego pola i głębokość łączonej końcówki, lub też brak jednej z tych dwóch rzeczy). Bierzemy teraz kolejno pola i starajmy się je przypisać zgodnie z tym, czego chce nasz algorytm. Mamy kilka przypadków:

1. Mamy pole, które zgodnie z algorytmem nie ma być przypisane niczemu w p, a teraz jest czemuś przypisane. W takim przypadku przypisanie to jest długości $< k-1$, inaczej nasz algorytm by je zrobił (tzn. jak już wspomniałem niekoniecznie akurat tego wierzchołka z tą końcówką, ale między wierzchołkiem i gaśnicą na takich głębokościach). Zatem możemy nasze przypisanie wydłużyć o 2 tak, żeby wystawało poza p. To jest to, czego trzeba — nie mamy już przypisania wewnątrz p.
2. Mamy pole X na głębokości i, które zgodnie z algorytmem ma być przypisane końcówce Y wystającej o i w górę (czyli na głębokości $k-i$), a teraz jest przypisane końcówce Y' wystającej o $i+1$ w górę. Jeśli końcówka Y nie jest niczemu przypisana, to nie ma problemu, po prostu przepinamy. Przypuśćmy, że końcówka Y jest połączona z pewnym polem X'. Istotne jest, że połączenie między X' i Y przechodzi przez korzeń p (bo Y jest w p, a w obrębie poddrzew p znajdują się jedynie połączenia zgodne z wymaganiami naszego algorytmu). Ponieważ Y' wystaje bardziej niż Y, to możemy połączyć X' z Y' (a X z Y) i dostajemy ścieżkę krótszą niż z X' do Y, czyli jest ok.
3. Nie może się zdarzyć, że mamy pole na głębokości i, które zgodnie z algorytmem ma być przypisane końcówce wystającej o $i+1$, a teraz jest przypisane końcówce wystającej o i w górę. Bowiem gdy jest możliwość stworzenia przypisania długości k nasz algorytm woli to od przypisania długości $k-1$.
4. Mamy pole x na głębokości i, które zgodnie z algorytmem ma być przypisane końcówce Y wystającej o i lub $i+1$ w górę, a teraz nie jest niczemu przypisane wewnątrz p. Oczywiście jest przypisane pewnej końcówce Y', ale to przypisanie wychodzi poza p. Jeśli Y nie jest z niczym połączona, to po prostu zamieniamy Y' na Y i jest ok. Przypuśćmy, że końcówka Y jest połączona z pewnym polem X'. Podobnie jak poprzednio wiemy, że połączenie to musi przechodzić przez korzeń p. Odległość od korzenia p do X' jest $\leqslant i+1$ (bo ścieżka z X' do Y jest $\leqslant k$), natomiast odległość od korzenia p do Y' jest $\leqslant k-i$ (bo ścieżka z X do Y' jest $\leqslant k$). Jeśli któraś z tych nierówności jest ostra, to możemy połączyć X z Y i X' z Y' i obie ścieżki są długości $\leqslant k$. Co jednak, gdy w obu miejscach mamy równości? Wówczas końcówka Y wystaje o $i+1$ w górę. Zauważmy, że wtedy oba pola X' i Y' są poza p. Gdyby bowiem na przykład pole X' było wewnątrz p, to nasz algorytm mógłby połączyć X' z Y ścieżką długości k i wolałby to zrobić niż łączyć X z Y ścieżką długości $k-1$. Podobnie, gdyby końcówka Y' była wewnątrz p, to nasz algorytm wolałby połączyć X z Y' ścieżką długości k, niż X z Y ścieżką długości $k-1$. Skoro więc oba X' i Y' są poza p, to najkrótsza ścieżka między nimi nie przechodzi przez korzeń p, lecz jest co najmniej o 2 krótsza niż (odległość od korzenia p do X')+(odległość od korzenia p do Y'), czyli jest długości co najwyżej $k-1$, czyli możemy dokonać zamiany połączeń.
5. Mamy pole na głębokości i, które zgodnie z algorytmem ma być przypisane końcówce wystającej o i lub $i+1$ w górę, a teraz jest przypisane końcówce wystającej o $\geqslant i+2$ w górę. Wówczas ścieżka przypisania

jest długości $\leqslant k-2$, możemy więc ją wydłużyć o 2, aby wystawała poza p, czyli żeby nie było przypisania wewnątrz p. Tym samym sprowadzamy sytuację do poprzedniego przypadku.

W końcu dochodzimy do korzenia. Wszystkie połączenia, które teraz utworzymy będą przechodzić przez korzeń drzewa. Z tego co pokazaliśmy do tej pory wynika, że to co teraz musimy zrobić to nie dodać za dużo gaśnic do korzenia. Niech S oznacza ilość pól nie jest powiązanych z żadną gaśnicą. Pierwsze co możemy zrobić to użyć maksymalnie dużo końcówek, które nam wystają z korzenia. Jeśli zużyjemy ich maksymalnie dużo to zminimalizujemy S, a co za tym idzie zminimalizujemy ilość dodanych gaśnic do korzenia. Otrzymujemy w ten sposób dosyć klasyczny problem. To co tak na prawdę robimy to przeglądamy po kolei pola, zaczynając od tych położonych najbliżej od korzenia. Każde takie pole próbujemy połączyć z najkrótszą wystarczającą końcówką (nie ma sensu używać dłuższej, bo może nam się później przydać). W ten sposób połączymy z gaśnicami kilka najbliższych korzeniowi pól. Każde inne połączenie można sprowadzić w oczywisty sposób do takiego. Skoro zużyliśmy największą możliwą ilość końcówek, to resztę pól trzeba połączyć z nowymi gaśnicami umieszczonymi w korzeniu.

Pliki: gas.cpp, gas1.pas,gas2.java

2 Rozwiązania zbyt wolne

2.1 Rozwiązanie $O(n^2)$

Wybieramy sobie dowolny wierzchołek r, który będzie korzeniem. Dla każdego wierzchołka v obliczamy sobie $d(v)$ - odległość v od korzenia w drzewie. Każdy wierzchołek jest w jednym z dwóch stanów: zabezpieczony lub niezabezpieczony. Zabezpieczone to takie, które mają już przydzieloną gaśnice. Niezabezpieczone to te, które jeszcze nie mają. Początkowo wszystkie wierzchołki są niezabezpieczone. Sortujemy wierzchołki po odległości od korzenia i przeglądamy je od tych najbardziej odległych do najbliższych. Teraz dla kolejnych wierzchołków v_i:

1. obliczamy $D(v_i)$ - zbiór wierzchołków niezabezpieczonych odległych o co najwyżej k od wierzchołka v_i.
2. sortujemy $D(v_i)$ po odległości od korzenia
3. dopóki w $D(v_i)$ istnieje wierzchołek odległy od korzenia o $d(v_i)+k$ bierzemy $min(s, |D(v_i)|)$ najbardziej odległych od korzenia wierzchołków w $D(v_i)$ i oznaczamy je jako zabezpieczone (trzeba zauważyć, że te wierzchołki są teraz zabezpieczone, czyli nie ma już ich w $D(v_i)$). Ta operacja jest związana z położeniem nowej gaśnicy w v_i.

Naszym wynikiem będzie *ilość dotychczas użytych gaśnic* $+\lceil$*ilość niezabezpieczonych wierzchołków* $/s\rceil$ Algorytm daje się zaimplementować w złożoności $O(n^2)$. Jest to algorytm zachłanny i jego poprawność można udowodnić podobnie jak poprawność rozwiązania autorskiego. To rozwiązanie powinno dostać 40-50 punktów.

Plik: gass2.cpp, gass3.pas

2.2 Rozwiązanie $O(n^n)$

Rozważamy wszystkie możliwe n-elementowe wariacje z powtórzeniami zbioru n-elementowego. Każda taka wariacja to przyporządkowuje każdemu wierzchołkowi v_i wierzchołek w_i. Oznacza to, że wierzchołek v_i jest zabezpieczony przez gaśnicę znajdującą się w wierzchołku w_i. Sprawdzamy, tylko te wariacje które są prawidłowe: dla każdego i wierzchołek w_i jest odległy od v_i o co najwyżej k. Dla każdej prawidłowej wariacji liczymy ile takie przyporządkowanie zużywa gaśnic i bierzemy z tego wszystkiego minimum, które jest wynikiem. Rozwiązanie to jest w oczywisty sposób poprawne, bo sprawdzamy wszystko. To rozwiązanie powinno dostawać maksymalnie 20 punktów.

Plik: gass0.cpp, gass1.pas

3 Rozwiązania niepoprawne

3.1 Rozwiązanie optymalne + zapomniałem long longów

To rozwiązanie powinno dostawać 90 punktów.

Plik: gasb0.cpp

3.2 Rozwiązanie prawie wzorcowe

Rozwiązanie nie wykonuje 5 kroku z rozwiązania autorskiego i otrzymuje 40 punktów.

Plik: gasb4.cpp

3.3 Rozwiązanie heurystyczne

Zwraca $\lceil n/s \rceil$.

Plik: gasb1.cpp

3.4 Rozwiązania złe ideowo 1

Dopóki jest jakiś wierzchołek niezabezpieczony, patrzymy na zbiory $D(v)$. Wybieramy wierzchołek v o maksymalnym $|D(v)|$. Ustawiamy w nim gaśnice i zabezpieczamy $min(|D(v), s|)$ najdalej oddalonych od korzenia wierzchołków w $D(v)$. Jeśli wszystkie wierzchołki są już zabezpieczone to patrzymy ile zużyliśmy gaśnic i to jest nasz wynik.

Plik: gasb2.cpp

3.5 Rozwiązania złe ideowo 2

Dopóki jest jakiś wierzchołek niezabezpieczony, patrzymy na zbiory $D(v)$. Wybieramy wierzchołek v o maksymalnym $|D(v)|$. Ustawiamy w nim gaśnice i zabezpieczamy $min(|D(v), s|)$ najmniej oddalonych od wierzchołka v wierzchołków w $D(v)$. Jeśli wszystkie wierzchołki są już zabezpieczone to patrzymy ile zużyliśmy gaśnic i to jest nasz wynik.

Plik: gasb3.cpp

4 Testy

Przygotowanych zostało 10 zestawów testowych (niektóre są zgrupowane). Dodatkowo zostały przygotowane 4 testy dla zawodników - jeden średniej wielkości. Testy dla zawodników nie są zbyt wymagające z oczywistych powodów.

Do wygenerowania stworzony został losowy generator drzew z parametrami, który rekurencyjne uruchamia się z korzenia przechodząc do jego synów, a w końcu do liści:
Parametry generatora

- s - s z zadania
- k - k z zadania
- *mnsons* - minimalna ilość synów w wierzchołku
- *mxsons* - maksymalna ilość synów w wierzchołku
- *sons* - maksymalna ilość wierzchołków w poddrzewie
- *deviation* - poziom niezrównoważenia drzewa $<$ 0..100)
- *length* - $<$ 0..100) im większe tym drzewo ma większą maksymalną głębokość (może być zniwelowane przez mxlevel)
- *mxlevel* - maksymalna głębokość drzewa
- *pathlevel* - poziom od którego, będą tworzone wierzchołki z jednym synem lub liściem -1 (brak)
- *shuffle* - czy losować etykiety w drzewie

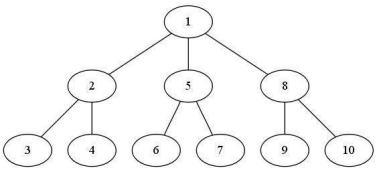
Od testu 5, testy z końcówką a to testy wydajnościowe. Testy $10b$,$10c$ to testy sprawdzające użycie long longów w rozwiązaniu autorskim. Punktacja poszczególnych rozwiązań jest zamieszczona w ich opisach. W opisach testów umieszczam parametry (w kolejności jak wyżej) z jakimi zostały one stworzone. Aby łatwiej było sobie wyobrazić jaki wpływ na kształt drzewa mają poszczególne parametry, umieszczam rysunki pierwszych testów.

Opis testów

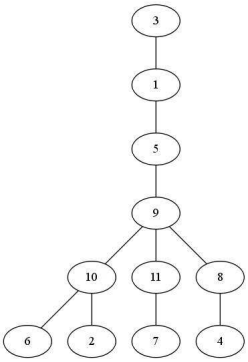
- `gas1ocen.IN` (0 sek.) parametry $1, 20, 1, 5, 10, 0, 0, 50, -1, prawda$
- `gas2ocen.IN` (0 sek.) parametry $2, 20, 5, 8, 30, 0, 0, 50, -1, prawda$
- `gas3ocen.IN` (0 sek.) parametry $2, 1, 1, 1, 50, 0, 0, 50, -1, fasz$
- `gas4ocen.IN` (0 sek.) parametry $10000, 3, 7000, 8000, 10000, 0, 0, 1, -1, fasz$
- `gas0.IN` (0 sek.) test przykładowy z treści zadania
- `gas1a.IN` (0 sek.) parametry $10, 3, 2, 4, 10, 0, 0, 4, -1, fasz$
- `gas1b.IN` (0 sek.) parametry $2, 2, 1, 3, 11, 40, 0, 100, -1, prawda$
- `gas2a.IN` (0 sek.) parametry $100000, 1, 1, 1000000, 20, 0, 0, 2, -1, fasz$
- `gas2b.IN` (0 sek.) parametry $3, 5, 1, 3, 25, 99, 0, 10, 6, prawda$
- `gas3a.IN` (0 sek.) parametry $10, 3, 3, 6, 200, 10, 0, 100, -1, prawda$
- `gas3b.IN` (0 sek.) parametry $200, 20, 1, 10, 400, 0, 0, 100, -1, prawda$
- `gas4a.IN` (0 sek.) parametry $7, 4, 1, 7, 1000, 60, 0, 30, 25, prawda$
- `gas4b.IN` (0 sek.) parametry $9, 18, 1, 5, 2000, 50, 3, 80, -1, prawda$
- `gas5a.IN` (0 sek.) parametry $3, 20, 3, 10, 6000, 10, 0, 30, 25, prawda$
- `gas5b.IN` (0 sek.) parametry $10, 16, 1, 2, 5000, 0, 3, 100, 95, fasz$
- `gas6a.IN` (0 sek.) parametry $7, 19, 3, 20, 10000, 35, 0, 60, -1, prawda$
- `gas6b.IN` (0 sek.) parametry $126, 7, 1, 50, 20000, 99, 5, 10000, -1, prawda$
- `gas7a.IN` (0 sek.) parametry $17, 20, 2, 100, 50000, 20, 0, 10000, -1, fasz$
- `gas7b.IN` (0 sek.) parametry $10000, 2, 1, 10, 50000, 70, 5, 10000, -1, fasz$
- `gas8a.IN` (0 sek.) parametry $31, 20, 10, 20, 70000, 20, 0, 10000, -1, prawda$
- `gas8b.IN` (0 sek.) parametry $40, 13, 1, 3, 70000, 80, 3, 50000, -1, fasz$
- `gas9a.IN` (0 sek.) parametry $131, 18, 1, 30, 90000, 0, 0, 50, -1, prawda$
- `gas9b.IN` (0 sek.) parametry $6000, 20, 1, 5, 100000, 50, 2, 20000, 10000, prawda$
- `gas10a.IN` (0 sek.) parametry $4, 20, 2, 2, 100000, 0, 0, 200, -1, fasz$
- `gas10b.IN` (0 sek.) parametry $100000, 2, 30000, 30000, 100000, 0, 0, 3, 1, fasz$
- `gas10c.IN` (0 sek.) parametry $100000, 2, 33000, 33000, 100000, 0, 0, 3, 1, prawda$

5 Rysunki testów

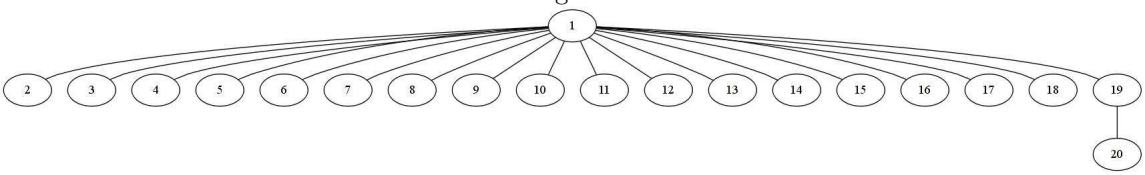
Test gas1a.in



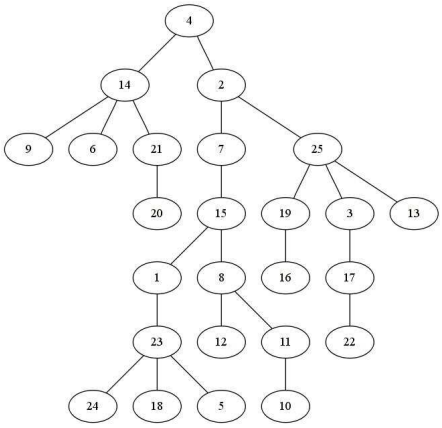
Test gas1b.in



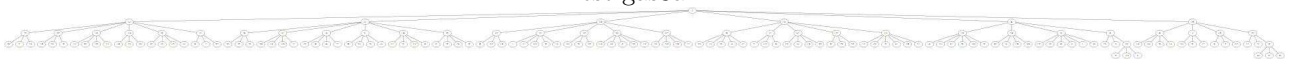
Test gas2a.in



Test gas2b.in



Test gas3a.in



Test gas3b.in

6 Limit czasowy i pamięciowy

Limity czasowe powinny być dobierane według opisów poszczególnych rozwiązań. Co do limitu pamięciowego to wystarczy 64MB. Problemem jest Java. Tutaj nie wiem od czego to zależy i jak jest to badane przez SIO. Możliwe, że trzeba będzie ustawić dla Javy inny limit.

7 Zmiany w treści zadania

- Ustalenie limitu pamięciowego na 64 MB.

Weryfikacja: gas/ppar009
Gaśnice

HISTORIA:
• v. 1.00: 2008.09.27, bgor, przygotowanie weryfikacji

dokument systemu SINOL 1.9.1

1 Alternatywne rozwiązanie

- Nie miałem pomysłu na alternatywne rozwiązanie tego zadania. Zaimplementowałem rozwiązanie wzorcowe `prog/gas3.cpp`w celu sprawdzenia poprawności odpowiedzi.

2 Zmiany

2.1 Zmiany w danych testowych

- Testy nie spełniała warunku, że dla każdej krawędzi (x, y) $x < y$. Poprawiłem generator danych wejściowych tak aby generował poprawne testy.

2.2 Inne Zmiany

- Poprawiłem weryfikator danych wejściowych tak aby sprawdzał dla każdej krawędzi (x, y) warunek $x < y$.
- Opracowujący dopracował dowód rozwiązania wzorcowego.

3 Lista wykonanych prac

- Poprawiłem weryfikator danych wejściowych.
- Poprawiłem generator danych wejściowych.
- Zaimplementowałem rozwiązania wolne w pascalu(`prog/gass1.pas`, `prog/gass3.pas`).
- Zaimplementowałem rozwiązanie niepoprawne `prog/gasb4.cpp`.
- Zweryfikowałem poprawność testów i odpowiedzi.

Dodatek B

Rozwiązanie wzorcowe zadania „Przyspieszenie algorytmu”

Poniżej zamieszczam rozwiązanie wzorcowe zadania „Przyspieszenie algorytmu” z XVI Olimpiady Informatycznej. Poniższy program cechuje się bardzo małą liczbą instrukcji wykonywanych w ciągu sekundy ze względu na nieliniowe dostępy do pamięci. Więcej informacji można znaleźć w punkcie 3.3.8 pracy.

```
/* Rozwiazanie wzorcowe do zadania PRZ (Przyspieszanie)
 * Autor: Adam Gawarkiewicz
 * Data: 28.08.2008
 * Zlozonosc obliczeniowa O(n*K).
 * Zlozonosc pamieciowa O(n + K).
 * (K - ilosc roznych znakow)
 */
#include <cstdio>
#include <cstring>

#define MAX_NUM 100
#define MAX_N 100000
#define X 0
#define Y 1

/* tablice przechowujace wierzcholki */
int s[2*MAX_N];
    // 0 lub 1 - ktorego slowa (x lub y) tyczy sie dany wierzcholek
int a[2*MAX_N], b[2*MAX_N];
    // konce przedzialu
int pKraw[2*MAX_N], pWart[2*MAX_N];
    // krawedz, wartosc przy przejsciu do prefiksu,
int sKraw[2*MAX_N], sWart[2*MAX_N];
    // krawedz, wartosc przy przejsciu do sufiksu,

inline bool rowneWierzch(int w1, int w2) {
    return pKraw[w1] == pKraw[w2] && pWart[w1] == pWart[w2]
        && sKraw[w1] == sKraw[w2] && sWart[w1] == sWart[w2];
}
```

```
int dl[2]; // dlugosc ciagu (n, m)
int tab1[2*MAX_N], tab2[2*MAX_N]; // do sortowania przez zliczanie
int ileWierzch;
int ile[2*MAX_N]; // potrzebne do sortowania przez zliczanie
int ileElem;
int maxWart; // maksymalna wartosc nadana przy numerowaniu wierzcholkow
int slowo[2][MAX_N];
int ilosc[2][MAX_NUM]; // ilosci danych liczb calkowitych w ciagu
int rozne[2]; // ilosc roznych znakow w ciagach
int najdluzszySufiks[2][2][MAX_N];
    // [poziom][slowo][i]: wartosci dla najdluzszych sufiksow slow [0..i]
int najdluzszyPrefiks[2][2][MAX_N];
    //[poziom][slowo][i]: wartosci dla najdluzszych prefiksow slow [i..n-1]
int poziom; // na ktorym poziomie akurat jestesmy - 0 lub 1

// operacje na zbiorze, wykorzystuja tablice ile
#define ROZMIAR ileElem
#define DODAJ(a) if (ile[a]++ == 0) ileElem++;
#define USUN(a) if (--ile[a] == 0) ileElem--;
#define ZAWIERA(a) (ile[a] != 0)

void tworzPoziom(int sl, int ileZnakow) {
    memset(ile, 0, sizeof(int)*MAX_NUM);
    ROZMIAR = 0;
    int poprzedniPoziom = (poziom^1);
    int i = 0, j = 0;
    int ostatnioDodany = 0;
    while (j < dl[sl]) {
        while (j < dl[sl] && (ROZMIAR < ileZnakow || ZAWIERA(slowo[sl][j]))) {
            if (!ZAWIERA(slowo[sl][j])) ostatnioDodany = slowo[sl][j];
            DODAJ(slowo[sl][j]);
            j++;
        }

        s[ileWierzch] = sl;
        a[ileWierzch] = i;
        b[ileWierzch] = j - 1;
        pWart[ileWierzch] = najdluzszyPrefiks[poprzedniPoziom][sl][i];
        sWart[ileWierzch] = najdluzszySufiks[poprzedniPoziom][sl][j - 1];
        pKraw[ileWierzch] = ostatnioDodany;
        while (ROZMIAR == ileZnakow) {
            sKraw[ileWierzch] = slowo[sl][i];
            USUN(slowo[sl][i]);
            i++;
        }
        ileWierzch++;
    }
}
```

```
int* sortujWierzcholki() {
    // sortuj wg pKraw, wynik zapisz w tab2
    memset(ile, 0, sizeof(int)*(maxWart+1));
    for (int i = 0; i < ileWierzch; i++)
        ile[pKraw[i]]++;
    for (int i = 1; i <= maxWart; i++)
        ile[i] += ile[i-1];
    for (int i = ileWierzch-1; i >= 0; i--)
        tab2[--ile[pKraw[i]]] = i;
    // sortuj tab2 wg pWart, wynik zapisz w tab1
    memset(ile, 0, sizeof(int)*(maxWart+1));
    for (int i = 0; i < ileWierzch; i++)
        ile[pWart[tab2[i]]]++;
    for (int i = 1; i <= maxWart; i++)
        ile[i] += ile[i-1];
    for (int i = ileWierzch-1; i >= 0; i--)
        tab1[--ile[pWart[tab2[i]]]] = tab2[i];
    // sortuj tab1 wg sKraw, wynik zapisz w tab2
    memset(ile, 0, sizeof(int)*(maxWart+1));
    for (int i = 0; i < ileWierzch; i++)
        ile[sKraw[tab1[i]]]++;
    for (int i = 1; i <= maxWart; i++)
        ile[i] += ile[i-1];
    for (int i = ileWierzch-1; i >= 0; i--)
        tab2[--ile[sKraw[tab1[i]]]] = tab1[i];
    // sortuj tab2 wg sWart, wynik zapisz w tab1
    memset(ile, 0, sizeof(int)*(maxWart+1));
    for (int i = 0; i < ileWierzch; i++)
        ile[sWart[tab2[i]]]++;
    for (int i = 1; i <= maxWart; i++)
        ile[i] += ile[i-1];
    for (int i = ileWierzch-1; i >= 0; i--)
        tab1[--ile[sWart[tab2[i]]]] = tab2[i];

    return tab1;
}

void numerujWierzcholki(int* tab) {
    int aktNumer = 0;
    najdluzszyPrefiks[poziom][s[tab[0]]][a[tab[0]]]
        = najdluzszySufiks[poziom][s[tab[0]]][b[tab[0]]] = aktNumer;
    for (int i = 1; i < ileWierzch; i++)
        najdluzszyPrefiks[poziom][s[tab[i]]][a[tab[i]]]
            = najdluzszySufiks[poziom][s[tab[i]]][b[tab[i]]]
            = (rowneWierzch(tab[i-1], tab[i]) ? aktNumer : ++aktNumer);
    maxWart = aktNumer > MAX_NUM ? aktNumer : MAX_NUM;
}
```

```
int main() {
    int k;
    scanf("%d", &k);
    while (k--) {
        memset(ilosc, 0, sizeof(ilosc));
        rozne[X] = rozne[Y] = 0;

        scanf("%d%d", &dl[0], &dl[1]);
        for (int s = X; s <= Y; s++)
            for (int i = 0; i < dl[s]; i++) {
                scanf("%d", &slowo[s][i]);
                slowo[s][i]--;
                if (!ilosc[s][slowo[s][i]]) rozne[s]++;
                ilosc[s][slowo[s][i]]++;
            }

        bool czyRozne = (rozne[X] != rozne[Y]);
        for (int i = 0; i < MAX_NUM; i++)
            if (!!ilosc[X][i] != !!ilosc[Y][i])
                czyRozne = true;

        if (czyRozne) { // W(x) != W(y)
            printf("0\n");
            continue;
        } else if (rozne[X]==1) { // |W(x)| = |W(y)| = 1
            printf("1\n");
            continue;
        }

        poziom = 0;
        for (int s = X; s <= Y; s++)
            for (int i = 0; i < dl[s]; i++)
                najdluzszyPrefiks[poziom][s][i]
                    = najdluzszySufiks[poziom][s][i] = slowo[s][i];
        maxWart = MAX_NUM;
        for (int ileZnakow = 2; ileZnakow <= rozne[X]; ileZnakow++) {
            poziom ^= 1;
            ileWierzch = 0;
            tworzPoziom(X, ileZnakow);
            tworzPoziom(Y, ileZnakow);
            numerujWierzcholki(sortujWierzcholki());
        }

        printf("%d\n", (najdluzszyPrefiks[poziom][X][0]
              == najdluzszyPrefiks[poziom][Y][0]));
    }
    return 0;
}
```

Bibliografia

[1] ACM: *Fact sheet — 28 dec 2008 — fourth edition.* `http://icpc.baylor.edu/About/FactSheet.pdf`, December 2008.

[2] Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986, ISBN 0-201-10088-6.

[3] AMD Inc.: *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, luty 2006.

[4] AMD Inc.: *AMD64 Architecture Programmer's Manual Volume 1: Application Programming*, wrzesień 2007.

[5] AMD Inc.: *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, wrzesień 2007.

[6] Deutsch, L. Peter and Allan M. Schiffman: *Efficient implementation of the Smalltalk-80 system.* In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 297–302, Salt Lake City, Utah, 1984. `citeseer.ist.psu.edu/deutsch84efficient.html`.

[7] Eranian, Stephane: *Perfmon2: A flexible performance monitoring interface for linux.* In *In Proc. of the 2006 Ottawa Linux Symposium*, pages 269–288, 2006.

[8] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*, czerwiec 2009.

[9] Intel Corporation: *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide Part 2*, June 2009.

[10] Iwanicki, Adam, Przemysława Kanarek, oraz Jakub Radoszewski (redaktorzy): *XV Olimpiada Informatyczna 2007/2008.* Olimpiada Informatyczna, 2008.

[11] Lattner, Chris and Vikram Adve: *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[12] Luk, Chi keung, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa, and Reddi Kim Hazelwood: *Pin: Building customized program analysis tools with dynamic instrumentation.* In *In Programming Language Design and Implementation*, pages 190–200. ACM Press, 2005.

[13] Nethercote, Nicholas and Julian Seward: *Valgrind: a framework for heavyweight dynamic binary instrumentation*. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 89–100. ACM, 2007. http://dx.doi.org/http://doi.acm.org/10.1145/1250734.1250746.

[14] Olimpiada Informatyczna: *Olimpiada Informatyczna — Podręcznik Użytkownika*, kwiecień 2009. Dokument wewnętrzny Olimiady Informatycznej.

Spis rysunków

Spis tablic

Spis listingów