Przedmowa

Dzień dobry. Mam na imię Kurt i jestem kodoholikiem.

Pisaniem kodu zajmuję się od ponad 35 lat. Nigdy nie pracowałem w Microsoft, Google, Facebook, Apple czy innej słynnej firmie. Jednak poza kilkoma krótkimi przerwami piszę kod każdego dnia. Ostatnie 20 lat poświęciłem prawie wyłącznie na rozwijanie kodu C++ oraz rozmawianie o języku C++ z innymi bardzo zdolnymi programistami. Tak wyglądają moje kwalifikacje do napisania książki o optymalizacji kodu C++. Posiadam również *spore* doświadczenie w pisaniu zwykłych tekstów, takich jak specyfikacje, komentarze, instrukcje, notatki czy wpisy na blogu (*http://oldhandsblog.blogspot.com/*). Ciągle dziwi mnie, że tylko połowa inteligentnych, kompetentnych programistów, z którymi miałem okazję współpracować, potrafi sklecić dwa poprawne zdania.

Jeden z moich ulubionych cytatów pochodzi z listu Izaaka Newtona, w którym pisze on: „Jeśli widzę dalej, to tylko dlatego, że stoję na ramionach gigantów". Ja także stoję na ramionach gigantów, a dokładniej przeczytałem ich książki: eleganckie i zwięzłe jak *Język ANSI C. Programowanie* Briana Kernighan i Dennisa Ritchie, mądre i postępowe jak seria *Effective C++* Scotta Meyersa, ambitne i rozszerzające horyzonty jak *Nowoczesne programowanie w C++* Andreia Alexandrescu, ostrożne i precyzyjne jak *The Annotated C++ Reference Manual* Bjarne Stroustrupa i Margaret Ellis. Przez długi czas nawet nie rozważałem możliwości zostania autorem, aż pewnego dnia, dość niespodziewanie, poczułem potrzebę napisania tej książki.

Dlaczego zdecydowałem się napisać książkę o podnoszeniu wydajności w języku C++?

Na początku XXI wieku język C++ znajdował się pod ostrzałem. Fani języka C wskazywali programy C++, których wydajność była gorsza niż teoretycznie równoważnego kodu napisanego w języku C. Słynne korporacje poświęcały ogromne budżety na reklamowanie własnych języków obiektowych, twierdząc, że język C++ jest zbyt trudny w użyciu i to ich narzędzia będą przyszłością sektora informatycznego. Uniwersytety decydowały się na nauczanie języka Java, ponieważ oferował on darmowy zestaw narzędzi. W efekcie duże firmy zaczęły odwracać się od technologii C++, wybierając coraz częściej język Java, C# lub PHP do implementowania serwisów internetowych oraz systemów operacyjnych. Był to niepewny czas dla osób przekonanych o użyteczności i ogromnych możliwościach języka C++.

Później jednak stała się rzecz niesłychana. Szybkość procesorów przestała rosnąć, mimo iż ich ilość pracy stale się zwiększała. Te same firmy zaczęły angażować programistów C++

w rozwiązywanie problemów ze skalowaniem. Koszt przepisania kodu do języka C++ był często mniejszy niż koszt zasilania centrów danych. Nagle język C++ wrócił do łask.

W odróżnieniu od innych języków powszechnie stosowanych na początku 2016 roku język C++ oferuje programistom pełne spektrum opcji implementacji, począwszy od zautomatyzowanego wsparcia bez konieczności nadzorowania po ścisłą kontrolę. Język C++ daje programistom ogromną swobodę w balansowaniu różnych aspektów wydajności. Tak wysoki stopień kontroli stwarza potencjał dla optymalizacji.

Mało książek opisuje optymalizację kodu C++. Dostępna jest solidna, lecz nieco już przestarzała książka *Efektywne programowanie w C++* Dova Bulki i Davida Mayhew. Jej autorzy mają doświadczenie podobne do mojego i doszli do wielu podobnych wniosków. Polecam ją czytelnikom, którzy chcieliby spojrzeć na omawiane w niniejszej książce problemy z nieco innej perspektywy. Dodatkowo, Scott Meyers (jak również wielu innych) dobrze i szczegółowo omówił problem eliminowania wywołań konstruktorów kopiujących.

Optymalizacja ma tak wiele aspektów, że można byłoby poświęcić jej nawet 10 książek. Dlatego musiałem wybrać te scenariusze, z którymi spotykam się najczęściej lub których optymalizacja może przynieść największe korzyści. Niektórzy czytelnicy posiadający doświadczenie w usprawnianiu kodu C++ mogą uznać, że pominąłem pewne sprawdzone i użyteczne strategie. Jednak ze względu na ograniczoną pojemność książki musiałem dokonać wyboru.

Zachęcam do przesyłania poprawek, komentarzy oraz ulubionych strategii optymalizacji na adres: *antelope_book@guntheroth.com*.

Uwielbiam rzemiosło, jakim jest rozwijanie oprogramowania. Mógłbym w nieskończoność ćwiczyć *kata* każdej nowej pętli lub interfejsu. Pisanie kodu podobnie jak pisanie wierszy to umiejętność tak ezoteryczna, sztuka tak osobista, że zrozumieć ją mogą tylko inni wtajemniczeni. W każdej elegancko napisanej funkcji znaleźć można piękno, podobnie jak w każdym trafnie użytym idiomie mądrość. Niestety na każdy wybitny programistyczny poemat, taki jak biblioteka Standard Template Library Stepanova, przypadają tysiące tomów nudnego, banalnego kodu.

Ta książka ma przede wszystkim zwrócić uwagę wszystkich czytelników na piękno zoptymalizowanego oprogramowania. Przeczytajcie ją i wykorzystajcie. Spójrzcie dalej!

Potencjalne problemy z kodem

Chociaż od ponad 20 lat zajmuję się pisaniem i optymalizowaniem kodu C++, większość kodu przedstawionego w tej książce została napisana specjalnie z myślą o tej książce i tak jak każdy nowy projekt z pewnością zawiera błędy. Za co z góry przepraszam.

Mam doświadczenie w programowaniu na platformę Windows, Linux oraz różne systemy wbudowane. Prezentowany kod zostały opracowany z myślą o platformie Windows i dlatego, podobnie jak cała książka, kładzie szczególny nacisk na aspekty związane z systemem Windows. Porady dotyczące optymalizacji kodu C++, choć zilustrowane przy użyciu narzędzia Visual Studio na platformie Windows, odnoszą się także do systemów

Linux, Mac OS X czy innych środowisk C++. Jednak parametry czasowe poszczególnych technik optymalizacji zależą od implementacji kompilatora i standardowej biblioteki oraz od procesora, na którym testowany jest kod. Optymalizacja stanowi eksperymentalną dziedzinę wiedzy. Przyjmowanie na wiarę porad dotyczących optymalizacji może pociągać za sobą niepożądane skutki.

Uzyskanie zgodności z różnymi kompilatorami oraz innymi systemami Unix i systemami wbudowanymi stanowi spore wyzwanie, dlatego przepraszam, jeśli przedstawiony kod nie kompiluje się w ulubionym systemie czytelnika. Ponieważ książka nie omawia zgodności między systemami, priorytetem było uzyskanie prostego kodu.

Przedstawiony poniżej sposób formatowania wcięć kodu z nawiasami klamrowymi nie jest moim ulubionym:

```
if (bool_condition) {
    controlled_statement();
}
```

Osobiście preferuję umieszczanie otwierających i zamykających nawiasów klamrowych w oddzielnych wierszach. Jednak powyższy sposób umożliwia umieszczenie większej liczby wierszy na jednej stronie druku, zatem zdecydowałem się na zastosowanie go w niniejszej książce.

Korzystanie z przykładowego kodu

Materiały pomocnicze (przykładowy kod, rozwiązania itp.) są dostępne do pobrania na stronie *www.guntheroth.com*.

Ta książka ma pomóc programistom w osiąganiu własnych celów. Dlatego wykorzystywanie przedstawionego przykładowego kodu w programach oraz dokumentacji jest zasadniczo dozwolone. Nie ma potrzeby kontaktowania się z nami w celu uzyskania zezwolenia, o ile nie planuje się reprodukcji znaczącej części kodu. Na przykład, napisanie programu wykorzystującego kilka fragmentów kodu z książki nie wymaga zezwolenia. Natomiast sprzedaż lub dystrybucja CD z przykładami wymaga zezwolenia. Udzielenie odpowiedzi poprzez zacytowanie tej książki i przykładowego kodu nie wymaga zezwolenia. Umieszczenie znaczącej części przykładów kodu z tej książki w dokumentacji własnego produktu wymaga zezwolenia.

Będziemy wdzięczni za wskazanie źródła, choć nie jest to wymagane. Przypis powinien zwykle zawierać nazwisko autora, tytuł, wydawnictwo oraz numer ISBN. Na przykład: „Kurt Guntheroth, *C++. Optymalizacja kodu*, O'Reilly/APN Promise, © 2016 Kurt Guntheroth, 978-83-7541-191-1".

W przypadku wątpliwości, czy planowane zastosowanie przykładowego kodu wykracza poza przedstawione powyżej zezwolenia, prosimy o skontaktowanie się z nami za pośrednictwem adresu e-mail: *permissions@oreilly.com*.

Konwencje użyte w tej książce

Oto konwencje typograficzne przyjęte w tej książce:

Zwykły tekst

: Odnosi się do elementów, tytułów i opcji menu oraz klawiszy skrótów (takich jak Alt czy Ctrl).

Kursywa

: Wyróżnia nowe terminy, adresy URL, adresy email, ścieżki, nazwy i rozszerzenia plików.

`Stała szerokość`

: Służy do przedstawiania fragmentów kodu programu, a także do odwoływania się w tekście do elementów programu, takich jak nazwy zmiennych, funkcji i baz danych, typy danych, zmienne środowiskowe, instrukcje czy słowa kluczowe.

Rozdział 1

Wprowadzenie do optymalizacji

Świat ma niesamowity apetyt na moc obliczeniową. Pewne programy wymagają ciągłego i szybkiego działania niezależnie od tego, czy kod jest uruchamiany na zegarku, telefonie, tablecie, stacji roboczej, superkomputerze czy ogólnoświatowej sieci centrów danych. Dlatego czasem nie wystarcza odpowiednie przekonwertowanie fajnego pomysłu w wiersze kodu. Nie wystarcza nawet wyeliminowanie wszystkich błędów poprzez wielokrotne przeczesanie kodu w poszukiwaniu usterek. Może się bowiem okazać, że aplikacja działa prawidłowo, ale zbyt wolno na sprzęcie, na jaki pozwala budżet klienta. Czasami mamy do dyspozycji jedynie mały procesor, ponieważ firma próbuje ograniczyć zużycie mocy. Czasem sukces w rywalizacji z konkurencją zależy od przepustowości lub liczby ramek na sekundę. Może się również zdarzyć, że po osiągnięciu skali niemal planetarnej firma zaczyna mieć obawy, że znacznie przyczynia się do ocieplenia globalnego. I tu pojawia się problem optymalizacji.

Ta książka poświęcona jest optymalizacji, a dokładniej optymalizowaniu programów C++ ze szczególnym naciskiem na powtarzalne aspekty działania kodu C++. Niektóre z opisanych w tej książce technik odnoszą się również do innych języków programowania, ale nie było to moim celem. Inne metody optymalizacji, które działają w kodzie C++, nie przynoszą żadnych pozytywnych efektów w innych językach lub nawet nie mogą być w nich stosowane.

Ta książka pokazuje, jak przekształcić poprawny kod zgodny z zaleceniami projektowymi języka C++ w poprawny kod, który nie tylko realizuje te zalecenia, ale dodatkowo działa szybciej i zużywa mniej zasobów na prawie każdym komputerze. Potrzeba optymalizacji często wynika z faktu, iż niektóre funkcje C++ stosowane bez zastanowienia spowalniają program i zużywają wiele zasobów. Uzyskane w ten sposób rozwiązanie, choć działa prawidłowo, jest krótkowzroczne. Często wynika to z faktu, iż programista posiada jedynie podstawową wiedzę o nowoczesnych mikroprocesorach lub nie uwzględnił kosztu różnych konstrukcji C++. Inne mechanizmy optymalizacji stają się możliwe, ponieważ język C++ zapewnia ścisłą kontrolę nad różnymi aspektami zarządzania pamięcią i kopiowania.

Ta książka *nie* opisuje, jak kodować pracochłonne procedury w assemblerze, jak zliczać cykle bądź ile instrukcji potrafi obsługiwać równolegle najnowszy produkt firmy

Intel. Niektórzy programiści od lat zajmują się jedną platformą (np. Xbox) i mają czas oraz potrzebę poznawać tę tajemną sztukę. Jednak zdecydowana większość programistów ma do czynienia z telefonami, tabletami lub komputerami, które mogą zawierać przeróżne mikroprocesory – także te, które nie zostały jeszcze zaprojektowane. Twórcy oprogramowania wbudowanego w produkty również napotykają różne procesory o odmiennej architekturze. Próba opanowania tajników tylu procesorów znacznie utrudniałaby programistom podejmowanie decyzji, a w skrajnych przypadkach mogłaby nawet doprowadzić ich do obłędu. Dlatego nie zalecam tego podejścia. Optymalizacja zależna od procesora nie ma sensu w większości aplikacji, które z założenia mogą być uruchamiane na różnych procesorach.

Ta książka *nie* wskazuje również najszybszego sposobu realizowania wybranych zadań w określonym systemie operacyjnym Windows, Linux, OS X czy systemach wbudowanych. Pokazuje, co można osiągnąć w języku C++, także przy użyciu standardowej biblioteki C++. Przeprowadzanie optymalizacji w sposób wykraczający poza możliwości języka C++ utrudnia kolegom z zespołu analizowanie i komentowanie zoptymalizowanego kodu. Nie należy pochopnie podejmować takiej decyzji.

Niniejsza książka uczy *jak* optymalizować. Próba stworzenia statycznego katalogu technik i funkcji skazana jest na porażkę, ponieważ ciągle opracowywane są nowe algorytmy i udostępniane nowe funkcje języka. Dlatego zdecydowałem się na przedstawienie kilku przykładów ilustrujących stopniowe udoskonalanie kodu, aby zaznajomić czytelników z procesem dostosowywania kodu i wykształcić w nich umiejętność dokonywania owocnej optymalizacji.

Ta książka pokazuje również, jak zoptymalizować proces programowania. Programiści świadomi kosztu wykonania opracowywanego kodu będą potrafili tworzyć kod, który od początku jest efektywny. Pisanie szybko działającego kodu zajmuje doświadczonemu programiście tyle samo czasu, ile pisanie kodu działającego powoli.

Dodatkowo, ta książka uczy, jak czynić cuda – jak po wprowadzeniu zmiany uzyskać od kolegów z zespołu reakcję typu: „Zaczęło działać niesamowicie szybko. Kto coś naprawił?”. Optymalizacja pomaga wzmocnić status programisty i zwiększyć dumę z wykonywanego zawodu.

Optymalizacja to część procesu rozwoju oprogramowania

Optymalizacja jest ściśle powiązana z pisaniem kodu. W tradycyjnym procesie rozwoju oprogramowania optymalizacja następowała po zakończeniu pracy nad kodem, w fazie integracji i testowania projektu, w której można analizować wydajność całego programu. W procesie Agile można poświecić optymalizacji jeden lub dwa sprinty po napisaniu

funkcji, która powinna spełniać pewne kryteria wydajności lub gdy konieczne jest zapewnienie określonej wydajności.

Optymalizacja ma na celu ulepszenie działania poprawnego programu tak, aby spełniał on wymagania klientów dotyczące prędkości, przepustowości, użycia pamięci, zużycia energii itp. Dlatego optymalizacja zajmuje w procesie rozwoju równie ważne miejsce co pisanie funkcji. Nieakceptowalnie niska wydajność stanowi taki sam problem dla użytkowników jak usterki czy brakujące funkcje.

Jedną z zasadniczych różnic między usuwaniem usterek a dostosowywaniem wydajności jest to, że wydajność stanowi zmienną ciągłą. Funkcja albo została zaimplementowana, albo nie. Usterka istnieje lub nie. Natomiast wydajność może być bardzo niska, bardzo wysoka lub gdzieś pośrodku. Ponadto optymalizacja stanowi proces iteracyjny – za każdym razem, gdy usunięta zostaje najwolniejsza część programu, wyłania się nowa najwolniejsza część.

Optymalizacja to w dużym stopniu sztuka eksperymentalna, która bardziej niż inne zadania programistyczne wymaga podejścia analitycznego. Aby przeprowadzać pomyślne optymalizacje, trzeba posiadać umiejętność obserwowania, formułowania testowalnych hipotez na podstawie tych obserwacji i przeprowadzania eksperymentów, które wiążą się z dokonywaniem pomiarów wspierających lub obalających postawione hipotezy. Doświadczeni programiści często uważają, że posiadają wiedzę i intuicję w zakresie tworzenia optymalnego kodu. Jednak ci, którzy nie testują regularnie swojej intuicji, często są w błędzie. Podczas pisania programów testowych do tej książki ja także kilka razy doświadczyłem sytuacji, w której wyniki testów były niezgodne z podpowiedziami mojej intuicji. Tematem tej książki są eksperymenty, a nie przeczucia.

Optymalizacja jest efektywna

Programiści mają problemy z przewidzeniem, jaki wpływ na ogólną wydajność dużego programu będą miały poszczególne podejmowane przez nich decyzje. Dlatego prawie każdy gotowy program zawiera obszary, które można znacząco zoptymalizować. Nawet kod opracowany przez doświadczone zespoły dysponujące dużą ilością czasu można często przyspieszyć o 30–100%. Presja czasowa lub brak doświadczenia mogą spowodować, że wydajność może zostać podniesiona nawet 3 do 10 razy. Dostosowując kod, trudno jest uzyskać jeszcze większą poprawę wydajności. Jednak wybór lepszego algorytmu lub struktury danych może oznaczać różnicę między funkcją gotową do wdrożenia a nieakceptowalną ze względu na niepraktycznie wolne działanie.

Optymalizacja jest OK

Wielu ekspertów stanowczo odradza dokonywanie optymalizacji. Zaleca, aby w ogóle z niej zrezygnować lub gdy to nieuniknione, zaczekać z nią do końca realizacji projektu i ograniczyć ją do minimum. Oto co słynny informatyk Donald Knuth powiedział o optymalizacji:

> Lepiej jest ignorować małe niewydajności przez, powiedzmy, 97 procent czasu: przedwczesna optymalizacja jest źródłem wszelkiego zła.
>
> Donald Knuth, *Structured Programming with go to Statements*, ACM Computing Surveys 6(4), Grudzień 1974, str. 268. CiteSeerX: 10.1.1.103.6084 (*http://bit.ly/knuth-1974*)

A oto opinia Williama A. Wulfa:

> W informatyce więcej grzechów zostało popełnionych w imię efektywności (czasem bez jej osiągnięcia) niż z jakiegokolwiek innego powodu – łącznie z głupotą.
>
> *A Case Against the GOTO. Proceedings of the 25th National ACM Conference*, 1972, str. 796

Obawa przed optymalizacją rozprzestrzeniła się do tego stopnia, że czasem nawet doświadczeni programiści wzdrygają się, gdy tylko rozmowa wkracza na temat dostosowywania wydajności. Moim zdaniem, opinia ta zbyt często jest cynicznie przytaczana jako usprawiedliwienie dla złych przyzwyczajeń programistycznych lub sposób uniknięcia krótkiej analizy, która mogłaby owocować uzyskaniem dużo szybszego kodu. Uważam, że bezkrytyczne akceptowanie tej porady doprowadziło do zmarnowania wielu cyklów procesora, czasu sfrustrowanych klientów i godzin pracy poświęconych na dostosowywanie kodu, który od początku powinien być bardziej efektywny.

Moja rada jest mniej dogmatyczna. Optymalizacja jest OK. Można nauczyć się idiomów efektywnego programowania i stosować je, nawet gdy nie ma pewności, że wydajność pisanego kodu ma kluczowe znaczenie. Te idiomy reprezentują dobry kod C++ i ich stosowanie nie spotka się z dezaprobatą kolegów z zespołu. Gdy ktoś zapyta, dlaczego nie napisaliśmy czegoś „prostego" i niewydajnego, można odpowiedzieć „Napisanie wydajnego kodu zajmuje tyle samo czasu co napisanie powolnego, marnotrawnego kodu. Dlaczego zatem miałbym celowo pisać nieefektywny kod?"

Natomiast nie ma sensu godzinami analizować wyboru najlepszego algorytmu, gdy jego wydajność może mieć niewielkie znaczenie. Nie warto poświęcać tygodni na pisanie w asemblerze kodu, którego czas działania *może* być istotny, tylko po to, aby zniweczyć cały wysiłek poprzez wywołanie kodu jako funkcji, uniemożliwiając kompilatorowi C++ jej wcielenie. Nie jest OK żądać, aby koledzy z zespołu napisali połowę programu w C, ponieważ „wszyscy wiedzą, że język C jest szybszy", gdy nie ma się pewności, że kod C

rzeczywiście działa szybciej ani że kod C++ działa powoli. Innymi słowy, wszystkie zalecenia dotyczące rozwoju programowania nadal mają zastosowanie. Optymalizacja nie usprawiedliwia łamania reguł.

Nie należy marnować czasu na optymalizację, gdy nie ma się pojęcia, gdzie tkwi źródło problemów. W rozdziale 3, „Mierzenie wydajności", wprowadzona została reguła 90/10, zgodnie z którą tylko około 10% kodu programu ma kluczowe znaczenie dla wydajności. Dlatego nie ma sensu modyfikowanie całego kodu programu w celu podniesienia jego efektywności. Ponieważ tylko 10% programu ma znaczący wpływ na wydajność, szanse przypadkowego wybrania odpowiedniego punktu wyjścia są niewielkie. W rozdziale 3 przedstawione zostały narzędzia, które pomagają w zidentyfikowaniu kluczowych miejsc w kodzie.

Gdy studiowałem na uniwersytecie, profesorowie ostrzegali nas, że optymalne algorytmy mają często wyższy koszt uruchomienia niż proste algorytmy. Dlatego należy je stosować tylko na dużych zbiorach danych. Chociaż reguła ta może sprawdzać się w przypadku pewnych ezoterycznych algorytmów, moje doświadczenie wskazuje, że optymalne algorytmy realizujące proste zadania wyszukiwania i sortowania wcale nie wymagają czasochłonnych przygotowań i przynoszą korzyści także w odniesieniu do niewielkich zbiorów danych.

Spotkałem się również z poradą, aby rozpocząć od zastosowania w programie najprostszego algorytmu i zoptymalizować go dopiero po stwierdzeniu, że program działa zbyt wolno. Ta porada pozwala na czynienie stopniowych postępów, jednak po zdobyciu pewnego doświadczenia implementowanie optymalnych operacji wyszukiwania lub sortowania zajmuje tyle samo czasu, co implementowanie wolniejszego algorytmu. Dlatego lepiej jest od razu zrealizować zadanie porządnie, dzięki czemu wystarczy debugować jeden algorytm.

Niektóre powszechnie panujące opinie są największym wrogiem procesu optymalizacji. Na przykład, „każdy wie", że optymalny algorytm sortowania ma złożoność O($n \log n$), gdzie n to rozmiar zbioru danych (podrozdział „Koszt czasowy algorytmów" w rozdziale 5 zawiera krótkie wprowadzenie do notacji wielkiego O i złożoności czasowej). Powszechna opinia przynosi tę korzyść, że powstrzymuje programistów przed uznaniem sortowania przez wstawianie o złożoności O(n^2) za optymalne rozwiązanie. Jednak stanowi problem, gdy zniechęca od zajrzenia do literatury i odkrycia, że szybszej działa metoda sortowania pozycyjnego o złożoności O($n \log_r n$) (gdzie r to pozycja lub liczba kubełków) lub że sortowanie flashsort osiąga jeszcze wyższą wydajność O(n) w przypadku losowo rozmieszczonych danych, lub że sortowanie szybkie, stanowiące zwykle punkt odniesienia dla pozostałych algorytmów, ma bardzo niską wydajność O(n^2) w najgorszym przypadku. Arystoteles powiedział kiedyś, że kobiety mają mniej zębów niż mężczyźni (*The History of Animals*, Book II, part 1 (*http://bit.ly/aristotle-animals*)) i ta opinia była powszechnie akceptowana przez 1500 lat, zanim ktoś wreszcie miał w sobie tyle ciekawości, aby policzyć zęby u kilku osób. Przeciwwagą dla powszechnie panujących opinii są metody naukowe przyjmujące postać eksperymentów. W rozdziale 3 omówione zostały narzędzia do mierzenia wydajności oprogramowania oraz eksperymenty sprawdzające skuteczność optymalizacji.

W świecie informatyki krąży również opinia, że optymalizacja nie ma znaczenia. Opinia ta bazuje na przekonaniu, że nawet jeśli kod działa obecnie powoli, szybkość procesorów stale wzrasta, a zatem za jakiś czas problem sam się rozwiąże. Podobnie jak inne niesprawdzone pogłoski, to stanowisko nigdy nie było do końca słuszne. Choć mogło sprawdzać się w latach 80-tych i 90-tych, gdy na rynku panowały standardowe komputery i niezależne aplikacje, a prędkość jednordzeniowych procesorów podwajała się co 18 miesięcy. Natomiast obecnie wielordzeniowe procesory mają wprawdzie coraz większą moc, ale wydajność poszczególnych rdzeni wzrasta powoli, a czasem nawet maleje. Ponadto dzisiejsze programy muszą działać również na platformach mobilnych, gdzie czas baterii i dyssypacja energii ograniczają tempo wykonywania instrukcji. Co więcej, choć z czasem pojawiają się nowi klienci dysponujący szybszym sprzętem, wydajność istniejącego sprzętu pozostaje taka sama. A jednocześnie ilość pracy rośnie. Jedyną szansą na zwiększenie szybkości działania oprogramowania u istniejących klientów jest optymalizacja nowszych wersji. Optymalizacja chroni program przed odejściem do lamusa.

Nanosekunda tu, nanosekunda tam

> Miliard tu, miliard tam i za chwilę mamy do czynienia z poważnymi kwotami.
>
> Cytat często błędnie przypisywany senatorowi Everettowi Dirksonowi (1898–1969), który odrzekał się od tych słów, choć przyznał, że zdarza mu się coś chlapnąć.

Komputery są niesamowicie szybkie. Potrafią wysyłać nową instrukcję szybciej niż co nanosekundę, czyli 10^{-9} sekundy! W obliczu takiej prędkości łatwo jest zbagatelizować znaczenie optymalizacji.

Problem polega na tym, że im szybszy procesor, tym szybsze nawarstwianie się zbędnych instrukcji. Jeśli 50% instrukcji wykonywanych przez program jest niepotrzebnych, program mógłby działać dwa razy szybciej po ich usunięciu, niezależnie od tempa wykonywania niepotrzebnych instrukcji.

Programiści twierdzący, że „wydajność nie ma znaczenia", odnoszą się często do pewnego typu aplikacji, podejmujących interakcję z użytkownikiem i działających na bardzo szybkich komputerach. Wydajność ma ogromne znaczenie w przypadku słabych wbudowanych i mobilnych procesorów z ograniczoną pamięcią, mocą lub szybkością. Jest również istotna na serwerach działających nieustannie pod dużym obciążeniem na wielkich maszynach. Innymi słowy, wydajność ma znaczenie w przypadku wszystkich aplikacji, które muszą radzić sobie z ograniczonymi zasobami (pamięcią, mocą, cyklami procesora). Wydajność ma również duże znaczenie, gdy zadanie jest na tyle duże, że warto rozproszyć je między wiele komputerów. W takiej sytuacji wydajność może oznaczać różnicę między kosztem utrzymywania stu serwerów lub instancji w chmurze bądź pięciuset, a nawet tysiąca.

W ciągu ostatnich 50 lat wydajność komputerów wzrosła o sześć rzędów wielkości, a mimo to nadal warto rozmawiać o optymalizacji. Jeśli ten trend się utrzyma, problem optymalizacji będzie istniał również w dalekiej przyszłości.

Podsumowanie strategii optymalizacji kodu C++

> Aresztować podejrzanych, tych co zwykle.
>
> Kpt. Louis Renault (Claude Rains), *Casablanca*, 1942

Różnorodność funkcji języka C++ zapewnia szerokie pole manewru podczas implementacji, począwszy od pełnej automatyzacji i swobody po ścisłą kontrolę nad wydajnością. Ta dowolność pozwala na dostosowywanie programów C++ do wymagań dotyczących wydajności.

Język C++ ma pewnych „zwykle podejrzanych" kandydatów do optymalizacji, takich jak wywołania funkcji, alokacja pamięci czy pętle. Oto lista metod podnoszenia wydajności programów C++, która stanowi jednocześnie konspekt niniejszej książki. Porady są szokująco proste i były już wcześniej publikowane. Jednak jak zwykle, diabeł tkwi w szczegółach. Przykłady i heurystyki przedstawione w tej książce ułatwią czytelnikom identyfikowanie możliwości optymalizacji.

Użyj lepszego kompilatora, lepiej używaj kompilatora

Kompilatory C++ to złożone artefakty. Różne kompilatory w różny sposób decydują o tym, jaki kod maszynowy ma zostać wygenerowany dla danych instrukcji C++. Widzą one różne możliwości optymalizacji. Produkują inne pliki wykonywalne dla tego samego kodu źródłowego. Gdy wydajność kodu ma kluczowe znaczenie, warto przetestować kilka kompilatorów w celu sprawdzenia, czy jeden z nich nie wygeneruje dla określonego kodu szybszego pliku wykonalnego.

Najważniejszą poradą dotyczącą wyboru kompilatora C++ jest *zastosowanie kompilatora zgodnego ze standardem C++11*. Standard C++11 implementuje referencje do r-wartości oraz semantykę przenoszenia danych, eliminując wiele operacji kopiowania, które były nieuniknione w poprzednich wersjach standardu C++ (semantyki przenoszenia danych zostaną omówione w „Implementowanie semantyki przenoszenia" w rozdziale 6).

Czasami *użycie lepszego kompilatora* polega na *lepszym użyciu kompilatora*. Na przykład, jeśli nasza aplikacja działa powoli, warto zajrzeć do opcji kompilatora, aby upewnić się, że optymalizator jest włączony. Mimo iż ta porada wydaje się oczywista, nadal jej udzielam, ponieważ wielokrotnie zdarzyło mi się usłyszeć w odpowiedzi, że faktycznie po włączeniu optymalizacji w kompilatorze kod zaczął działać dużo szybciej. W wielu sytuacjach to wystarczy. Kompilator może kilkakrotnie przyspieszyć działanie programu, wystarczy ładnie go o to poprosić.

Większość kompilatorów domyślnie nie włącza żadnego mechanizmu optymalizacji. Czasy kompilacji są nieco krótsze bez przeprowadzania optymalizacji. Miało to większe znaczenie w latach 90-tych, jednak dzisiejsze kompilatory i komputery są tak szybkie, że dodatkowy koszt nie powinien stanowić problemu. Wyłączenie optymalizacji ułatwia również debugowanie, ponieważ instrukcje są wykonywane w kolejności zgodnej z tą zdefiniowaną w kodzie źródłowym. Natomiast optymalizator może przenosić kod z pętli, usuwać niektóre wywołania funkcji, a nawet pewne wybrane zmienne. Niektóre kompilatory w ogóle nie emitują symboli debugowania, gdy włączona jest optymalizacja. Inne są bardziej hojne, ale zrozumienie działania programu poprzez analizowanie wykonania w debugerze może przysporzyć pewnych trudności. Wiele kompilatorów pozwala na włączanie lub wyłączanie poszczególnych optymalizacji w kompilacji debugowania bez dużego wpływu na debugowanie. Samo włączenie opcji wcielania funkcji może mieć duży wpływ na program C++, ponieważ w języku C++ zalecanym stylem jest pisanie wielu krótkich funkcji składowych, które służą do uzyskiwania dostępu do zmiennych danej klasy.

Dokumentacja dostarczana z kompilatorem C++ zawiera szczegółowy opis dostępnych flag i dyrektyw optymalizacji. Ta dokumentacja pełni podobną rolę do instrukcji obsługi dostarczanej z każdym nowym samochodem. Można po prostu wsiąść do samochodu i zacząć nim jeździć bez zapoznania się z instrukcją, ale zawiera ona wiele cennych informacji, które mogą pomóc w efektywniejszym wykorzystywaniu tego dużego, skomplikowanego instrumentu.

Jeśli mamy to szczęście, że zajmujemy się rozwojem oprogramowania dla architektury x86 na platformie Windows lub Linux, mamy do wyboru wiele doskonałych i stale ulepszanych kompilatorów. Firma Microsoft opublikowała trzy wersje programu Visual C++ w ciągu pięciu lat poprzedzających napisanie tej książki. GCC publikuje więcej niż jedną wersję rocznie.

W czasie powstawiania tej książki (w pierwszej połowie 2016 roku) większość osób zgadzała się, że kompilator C++ firmy Intel generuje najzwięźlejszy kod na platformy Linux oraz Windows, kompilator C++ GNU GCC charakteryzuje się niższą wydajnością, ale doskonałą zgodnością ze standardami, natomiast kompilator Visual C++ firmy Microsoft plasuje się gdzieś między nimi. Chciałbym ułatwić czytelnikom podejmowanie decyzji, udostępniając prosty wykres wskazujący, że kompilator Intel C++ generuje kod o określony procent szybszy niż GCC, jednak to zależy od kodu oraz tego, kto ostatnio opublikował najnowszą udoskonaloną wersję. Kompilator Intel C++ kosztuje ponad tysiąc dolarów, ale oferuje darmową 30-dniową wersję próbną. Istnieją darmowe wersje kompilatora Visual C++ (Express). Natomiast na platformie Linux kompilator GCC jest zawsze darmowy. Dzięki temu można tanio przeprowadzić prosty eksperyment, testując każdy z kompilatorów na swoim kodzie i sprawdzając, czy któryś z nich pozwala uzyskać wyższą wydajność.

Historia pewnej optymalizacji

Dawno temu, w czasach 8-calowych dyskietek i 1 MHz-owych procesorów, pewien programista zaprojektował program do zarządzania stacjami radiowymi. Jednym z zadań tego programu było sporządzanie posortowanej listy piosenek nadawanych każdego dnia. Problem polegał na tym, że sortowanie danych zebranych jednego dnia zajmowało 27 godzin, co oczywiście było nie do zaakceptowania. Programista włożył wiele wysiłku w to, aby przyspieszyć działanie tej operacji. Przeprowadził inżynierię odwrotną na komputerze i za pomocą nieudokumentowanych metod włamał się do mikroprogramu. W mikrokodzie zakodował sortowanie w pamięci, obniżając czas wykonania do nadal nieakceptowalnych 17 godzin. Zrozpaczony zadzwonił z prośbą o pomoc do producenta komputera, dla którego pracowałem.

Spytałem tego programistę, jakiego algorytmu sortowania używa. Odpowiedział: „Sortowania przez scalanie". Sortowanie przez scalanie należy do rodziny optymalnych algorytmów sortowania przez porównanie. Na pytanie, ile pozycji zawiera sortowana lista, usłyszałem odpowiedź: „Kilka tysięcy". To nie miało sensu. Wykorzystywany system powinien poradzić sobie z sortowaniem tych danych w niecałą godzinę.

Wpadłem na pomysł, aby poprosić programistę o szczegółowe opisanie zastosowanego algorytmu sortowania. Nie pamiętam już jego dokładnych słów, ale z opisu wynikało, że programista zaimplementował sortowanie przez wstawianie. Sortowanie przez wstawianie to zły wybór, ponieważ czas jego wykonania jest proporcjonalny do kwadratu liczby sortowanych pozycji (patrz „Koszt czasowy algorytmów sortowania" w rozdziale 5). Programista wiedział, że istnieje coś takiego jak sortowanie przez scalanie i zdołał opisać zaimplementowaną przez siebie operację sortowania przez wstawianie, używając słów „scalanie" i „sortowanie".

Zaimplementowałem dla tego klienta standardową procedurę sortowania przez scalanie, która posortowała dane w 45 minut.

Użyj lepszych algorytmów

Największe korzyści w procesie optymalizacji przynosi wybranie optymalnego algorytmu. Optymalizacja może w znaczący sposób podnieść wydajność programu. Może rozruszać wolno działający kod, podobnie jak modernizacja komputera przyspiesza działanie aplikacji. Niestety podobnie jak modernizacja komputera, większość procesów optymalizacji podnosi wydajność tylko do pewnego stopnia, z reguły od 30% do 100%. W najpomyślniejszym scenariuszu można potroić wydajność. Jednak rzadko można osiągnąć rewolucyjną poprawę wydajności, bez znalezienia dużo efektywniejszego algorytmu.

Nie ma sensu podejmować heroicznej walki w celu zoptymalizowania złego algorytmu. Poznanie i stosowanie optymalnych algorytmów do wyszukiwania oraz sortowania

stanowi najprostszą drogę do uzyskania optymalnego kodu. Nieefektywna procedura wyszukiwania lub sortowania może mieć ogromny wpływ na czas działania całego programu. Dostosowywanie kodu pozwala na skrócenie czasu o stały współczynnik. Zastosowanie bardziej optymalnego algorytmu pozwala na skrócenie czasu wykonania o współczynnik, którego wartość wzrasta wraz z rozmiarem zbioru danych. Nawet w przypadku małych zbiorów danych zawierających kilkanaście elementów optymalna operacja wyszukiwania lub sortowania pozwala zaoszczędzić wiele czasu, jeśli dane są często przeszukiwane. W rozdziale 5, „Optymalizowanie algorytmów", przedstawione zostaną pewne wskazówki pomocne przy wybieraniu optymalnych algorytmów.

Istnieje wiele różnych okazji do zastosowania optymalnych algorytmów, począwszy od prostych pojedynczych obliczeń po zwięzłe funkcje wyszukiwania słowa kluczowego, złożone struktury danych oraz ogromne programy. Temat ten został omówiony w wielu doskonałych książkach. Wielu programistów poświęciło swoje kariery na studiowanie tego problemu. Żałuję, że w tej książce mogę jedynie nadmienić kwestię optymalnych algorytmów.

W podpunkcie „Wzorce optymalizacji" w rozdziale 5 omówione zostały wybrane kluczowe techniki podnoszenia wydajności, między innymi *wstępne obliczanie* (przenoszenie obliczeń z czasu uruchomienia do czasu konsolidacji, kompilowania lub projektowania), *opóźnione obliczanie* (przenoszenie obliczeń w miejsce, w którym wykorzystywane tylko czasami wyniki są naprawdę potrzebne) oraz *buforowanie* (zapisywanie i ponowne wykorzystywanie kosztownych obliczeń). W rozdziale 7, „Optymalizowanie aktywnych instrukcji", przedstawione zostały przykładowe zastosowania tych technik.

Użyj lepszych bibliotek

Standardowe biblioteki szablonów i środowiska uruchomieniowego, które są dostarczane wraz z kompilatorem języka C++, muszą być łatwe w utrzymaniu, uniwersalne i niezawodne. Jednak, co może zaskoczyć niektórych programistów, niekoniecznie są one zaprojektowane z myślą o szybkości. Jeszcze większym zaskoczeniem może okazać się fakt, iż mimo 30-letniej historii języka C++, biblioteki dostarczane wraz z komercyjnymi kompilatorami C++ nadal zawierają usterki i nie zawsze są zgodne z aktualnym standardem C++ lub nawet ze standardem obowiązującym w momencie publikowania kompilatora. To utrudnia dokonywanie pomiarów lub rekomendowanie metod optymalizacji i sprawia, że zdobyte doświadczenie w optymalizacji niekoniecznie odnosi się do innych środowisk. Problemy te zostaną omówione w rozdziale 8, „Zastosowanie lepszych bibliotek".

Opanowanie standardowej biblioteki C++ stanowi kluczową umiejętność programisty zajmującego się optymalizacją. Ta książka zawiera zalecenia dotyczące algorytmów wyszukiwania i sortowania (rozdział 9), optymalne idiomy stosowania klas kontenera (rozdział 10), operacji we/wy (rozdział 11), równoległości (rozdział 12) oraz zarządzania pamięcią (rozdział 13).

Dostępne są biblioteki typu open source oferujące ważne funkcje, takie jak zarządzanie pamięcią (patrz „Wysoko wydajne menedżery pamięci" w rozdziale 13), które zostały

zaimplementowane w sposób zapewniający szybsze działanie i szersze możliwości niż domyślna biblioteka środowiska uruchomieniowego C++. Zaletą tych alternatywnych bibliotek jest to, że z reguły można łatwo dodać je do istniejącego projektu i uzyskać natychmiastową poprawę szybkości.

Strony internetowe projektu Boost (*http://www.boost.org/*), Google Code (*https://code.google.com/*) i inne oferują wiele bibliotek służących do realizowania m.in. operacji we/we, obsługi okien, obsługi ciągów (patrz podpunkt „Użyj nowatorskiej implementacji ciągów" w rozdziale 4) i obsługi równoległości (patrz „Biblioteki wspierające równoległość" w rozdziale 12). Nie pełnią one roli bezpośrednich zamienników standardowych bibliotek, lecz oferują większą wydajność i dodatkowe funkcje. Aczkolwiek poprawa wydajności wynika częściowo z ustalenia innych priorytetów niż te towarzyszące tworzeniu standardowej biblioteki.

Dodatkowo można opracować specyficzną dla projektu wersję biblioteki, która uzyskuje większą wydajność kosztem pewnych gwarancji bezpieczeństwa i niezawodności, jakie daje standardowa biblioteka. Te zagadnienia zostaną omówione w rozdziale 8.

Wywołania funkcji są kosztowne pod wieloma względami (patrz „Koszt wywołań funkcji" w rozdziale 7). Dobre interfejsy API biblioteki funkcji oferują funkcje, które odzwierciedlają idiomy użycia tych API, co ogranicza liczbę wywołań kluczowych funkcji. Na przykład, interfejs API, który pobiera znak i oferuje jedynie funkcję `get_char()`, wymaga od użytkownika wywołania funkcji za każdym razem, gdy potrzebny jest znak. Gdyby interfejs API oferował także funkcję `get_buffer()`, można byłoby wyeliminować koszt wywoływania funkcji dla każdego znaku.

Biblioteki funkcji i klas stanowią dobry sposób na ukrycie złożoności, która czasem towarzyszy wysoko zoptymalizowanym programom. Biblioteki powinny równoważyć koszt ich wywoływania, realizując operacje w maksymalnie efektywny sposób. Funkcje biblioteki zajmują zwykle dolne pozycje w głęboko zagnieżdżonych łańcuchach wywołania, gdzie efekty podniesionej wydajności są potęgowane.

Zredukuj alokację pamięci i kopiowanie

Eliminowanie wywołań menedżera pamięci stanowi tak efektywną technikę, że programiści mogą odnosić sukcesy w zakresie optymalizacji, stosując tylko ten jeden trik. Choć koszt większości funkcji języka C++ ogranicza się do maksymalnie kilku instrukcji, koszt każdego wywołania menedżera pamięci jest mierzony w tysiącach instrukcji.

Ponieważ ciągi stanowią tak istotną (i kosztowną) część wielu programów C++, cały rozdział został poświęcony analizie przypadku optymalizacji ich użycia. Rozdział 4 wprowadza i objaśnia wiele technik optymalizacji związanych z przetwarzaniem ciągów. Rozdział 6, „Optymalizacja zmiennych dynamicznych", został poświęcony redukowaniu kosztu dynamicznej alokacji pamięci bez rezygnowania z idiomów programistycznych C++, takich jak ciągi i kontenery standardowej biblioteki.

Jedno wywołanie funkcji kopiującej bufor może zużywać nawet tysiące cyklów. Dlatego zmniejszanie liczby operacji kopiowania stanowi oczywisty sposób optymalizowania

kodu. Wiele operacji kopiowania wiąże się z alokowaniem pamięci, a zatem rozwiązanie jednego problemu eliminuje często drugi problem. Inne obszary związane z nasilonym kopiowaniem to konstruktory, operatory przypisania oraz operacje wejścia/wyjścia. Zagadnienie to zostanie omówione w rozdziale 6.

Usuń obliczenia

Koszt pojedynczych instrukcji C++, za wyjątkiem wywołań alokacji i funkcji, jest zwykle niewielki. Problem pojawia się jednak, gdy ten sam kod zostaje wykonany miliony razy w pętli lub za każdym razem, gdy program przetwarza zdarzenie. Większość programów zawiera przynajmniej jedną główną pętlę do przetwarzania zdarzeń i przynajmniej jedną funkcję przetwarzającą znaki. Zidentyfikowanie i zoptymalizowanie tych pętli prawie zawsze przynosi korzyści. Z rozdziału 7 można się dowiedzieć, jak odnaleźć najczęściej wykonywany kod. Prawie zawsze jest on umieszczony w pętli.

Książki i artykuły poświęcone optymalizacji przedstawiają mnóstwo technik efektywnego stosowania poszczególnych instrukcji C++. Wielu programistów uważa, że ich opanowanie stanowi klucz do skutecznej optymalizacji. Problem polega na tym, że, o ile kod nie jest ekstremalnie aktywny (często wykonywany), usunięcie z niego jednej lub dwóch operacji dostępu do pamięci będzie miało niewielki wpływ na ogólną wydajność programu. W rozdziale 3 przedstawione zostały metody sprawdzania, które części programu są często wykonywane w celu zredukowania liczby obliczeń wykonywanych w tych miejscach.

W rzeczywistości nowoczesne kompilatory C++ świetnie radzą sobie z identyfikowaniem tego typu potencjalnych, lokalnych ulepszeń. Dlatego nie należy przesadzać i dostosowywać ogromnej ilości kodu, zastępując np. każde wystąpienie `i++` wyrażeniem `++i`, odwijając wszystkie pętle lub z zapałem tłumacząc każdemu koledze z zespołu, na czym dokładnie polega mechanizm Duffa i gdzie tkwi jego geniusz. Mimo to przedstawię wprowadzenie do ogromu różnych metod w rozdziale 7.

Użyj lepszych struktur danych

Wybranie odpowiedniejszej struktury danych ma ogromny wpływ na wydajność. Wynika to częściowo z faktu, iż czas wykonania w algorytmach wstawiania, iteracji, sortowania i odczytywania danych zależy od struktury danych. Ponadto, różne struktury danych w różny sposób wykorzystują menedżera pamięci – między innymi dlatego, że niektóre z nich charakteryzują się wysoką lokalnością odwołań do pamięci podręcznej. W rozdziale 10 omówione zostaną wydajność, działanie oraz wady i zalety poszczególnych struktur danych dostępnych w standardowej bibliotece C++. Rozdział 9 przedstawia zastosowania algorytmów ze standardowej biblioteki do implementacji tabelarycznych struktur danych, a także prostych wektorów i tablic C.

Zwiększ równoległość

Wiele programów musi oczekiwać na zakończenie operacji, które są realizowane w uciążliwej fizycznej rzeczywistości. Trzeba czekać na odczytanie plików z mechanicznych dysków, na pobranie stron z Internetu czy na to, aż powolne palce użytkowników nacisną mechaniczne klawisze. Każda sytuacja, gdy postęp programu jest blokowany przez oczekiwanie na tego typu zdarzenie, reprezentuje zmarnowaną szansę na przeprowadzenie innych obliczeń.

Nowoczesne komputery dysponują więcej niż jednym rdzeniem procesora, na którym można wykonywać instrukcje. Jeśli praca zostaje rozdzielona między różne procesory, zadanie można zrealizować szybciej.

Wraz z możliwością równoległego wykonania pojawiły się narzędzia służące do synchronizowania równoległych wątków tak, aby mogły one dzielić się danymi. Rozdział 12 opisuje pewne czynniki, które warto wziąć po uwagę, aby w efektywny sposób synchronizować wątki.

Zoptymalizuj zarządzanie pamięcią

Menedżer pamięci, składnik biblioteki środowiska uruchomieniowego C++, który zarządza przydzielaniem pamięci dynamicznej, stanowi często wykonywany kod w wielu programach C++. Język C++ oferuje szczegółowy interfejs API do zarządzania pamięcią, choć większość programistów nigdy z niego nie korzystała. W rozdziale 13 przedstawione zostały niektóre techniki efektywniejszego zarządzania pamięcią.

Podsumowanie

Ta książka pomaga programistom w zidentyfikowaniu i wykorzystaniu następujących okazji do podniesienia wydajności kodu:

- *Użycie lepszych kompilatorów i włączenie optymalizatora.*
- *Zastosowanie optymalnych algorytmów.*
- *Użycie lepszych bibliotek i lepsze użycie bibliotek.*
- *Obniżenie przydziału danych.*
- *Redukcja kopiowania.*
- *Usunięcie obliczeń.*
- *Wykorzystanie optymalnych struktur danych.*
- *Zwiększenie równoległości.*
- *Optymalizacja zarządzania pamięcią.*

Jak wspomniałem wcześniej, diabeł tkwi w szczegółach. A zatem pora zabrać się do pracy.

Rozdział 2

Wpływ działania komputera na optymalizację

> Kłamstwa, opowiadanie pięknych nieprawdziwych historii, to główny cel sztuki.
>
> Oscar Wilde, „The Decay of Lying", *Intentions* (1891)

Ten rozdział ma na celu zaprezentowanie absolutnie niezbędnych informacji o sprzęcie komputerowym, które mają wpływ na optymalizację. To uchroni czytelników przed przerażającą wizją lektury ponad 600-stronicowych instrukcji procesorów. Przedstawiony zostanie tylko pobieżny przegląd architektury procesora, który powinien jednak pozwolić na wyodrębnienie pewnych heurystyk pomocnych podczas optymalizacji. Bardzo niecierpliwi czytelnicy mogą pominąć ten rozdział i powrócić do niego dopiero wtedy, gdy napotkają odwołania do niego w dalszej części książki. Niemniej przedstawione w tym rozdziale informacje są istotne i pomocne.

Wykorzystywane obecnie urządzenia mikroprocesorowe znacznie różnią się od siebie. Na jednym końcu skali znajdują się super tanie urządzenia wbudowane z zaledwie kilkoma tysiącami bramek i częstotliwością zegara poniżej 1 MHz, a na drugim urządzenia komputerowe z miliardami bramek i gigahercowym zegarami. Komputery typu mainframe mogą osiągać rozmiar pokoju, składać się z tysięcy niezależnych jednostek wykonania i pobierać ilość prądu, która wystarczyłaby do oświetlenia niewielkiego miasta. Z pozoru może się wydawać, że różnice pomiędzy tymi urządzeniami są zbyt wielkie, aby mogło je cokolwiek łączyć. Jednak w rzeczywistości istnieją pewne podobieństwa, z których warto zdawać sobie sprawę. W końcu gdyby nie te podobieństwa, nie można byłoby kompilować kodu C++ na tyle różnych procesorów.

Wszystkie powszechnie wykorzystywane komputery wykonują instrukcje przechowywane w pamięci. Instrukcje realizują operacje na danych, które również są przechowywane w pamięci. Pamięć dzieli się na wiele małych *słów*, a każde z nich składa się z kilku bitów. Kilka najważniejszych słów pamięci to *rejestry*, które są bezpośrednio nazwane w instrukcjach maszynowych. Do nazywania większości słów służy *adres* numeryczny. Określony rejestr na każdym komputerze zawiera adres następnej instrukcji do wykonania. Gdybyśmy porównali pamięć do książki, *adres wykonania* przypominałby palec

wskazujący następne słowo do odczytania. *Jednostka wykonania* (zwana również procesorem, rdzeniem, CPU, komputerem itp.) odczytuje strumień instrukcji z pamięci i podejmuje odpowiednie działania. Instrukcje mówią jednostce wykonania, które dane mają zostać odczytane (pobrane, załadowane) z pamięci, jakie operacje mają zostać wykonane na danych i gdzie w pamięci ma zostać zapisany (przechowany, utrwalony) wynik. Komputer składa się z urządzeń, które są podporządkowane prawom fizyki. Odczytanie i zapisanie każdego adresu w pamięci zajmuje pewną, niezerową ilość czasu, podobnie jak wykonanie instrukcji na danych.

Ten podstawowy mechanizm znany jest każdemu studentowi pierwszego roku informatyki. Jednak rodzina architektur komputerowych ewoluuje w wielu różnych kierunkach. Ze względu na ogromne różnice architekturalne poszczególnych komputerów trudno jest sformułować konkretne matematyczne reguły dotyczące działania sprzętu. Nowoczesne procesory stosują tak wiele różnych, powiązanych ze sobą rozwiązań w celu przyspieszenia procesu wykonywania instrukcji, że pomiary czasu wykonania instrukcji stały się praktycznie niedeterministyczne. Sytuację dodatkowo komplikuje fakt, iż programiści często nie są nawet pewni, na jakim procesorze będzie uruchamiany ich kod. W konsekwencji można jedynie osiągnąć wyniki o charakterze heurystycznym.

Nieprawdziwe przekonania języka C++ o komputerach

Oczywiście, program C++ przynajmniej udaje, że wierzy w pewną wersję uproszczonego modelu komputera. Istnieje praktycznie nieograniczona pamięć adresowalna w bajtach o rozmiarze `char`. Istnieje specjalny adres `nullptr`, inny niż którykolwiek prawidłowy adres w pamięci. Liczba całkowita 0 jest konwertowana do `nullptr`, choć `nullptr` niekoniecznie znajduje się pod adresem 0. Istnieje jeden konceptualny adres wykonania wskazujący aktualnie wykonywaną instrukcję kodu źródłowego. Instrukcje są wykonywane w kolejności, w której zostały napisane, za wyjątkiem działania instrukcji sterowania C++.

C++ wie, że komputery są w rzeczywistości bardziej skomplikowane niż ten prosty model. Oferuje parę sposobów zaglądania do środka tej wspaniałej maszyny:

- Język C++ musi jedynie zachowywać się, „jak gdyby" instrukcje były kolejno wykonywane. Kompilator C++ i sam komputer mogą zmienić kolejność wykonania, aby przyspieszyć działanie programu, o ile wynik obliczeń pozostaje niezmieniony.
- Od standardu C++11, język C++ nie wierzy już w obecność tylko jednego adresu wykonania. Standardowa biblioteka C++ wspiera obecnie rozpoczynanie i zatrzymywanie wątków oraz synchronizowanie dostępu do pamięci między wątkami. Przed standardem C++11, programiści ukrywali przed kompilatorem C++ prawdę o wątkach, co prowadziło czasem do powstawania problemów trudnych do debugowania.
- Niektóre adresy pamięci mogą być rejestrami urządzeń, a nie zwykłej pamięci. Wartości tych adresów mogą zmienić się w czasie między dwoma kolejnymi odczytami określonej lokalizacji przez ten sam wątek, co sygnalizuje wystąpienie pewnej

sprzętowej zmiany. Tego typu lokalizacje są opisywane w kodzie C++ jako `volatile`. Zadeklarowanie zmiennej jako `volatile` oznacza, że kompilator musi pobrać nową kopię zmiennej za każdym razem, gdy zostaje ona użyta, zamiast optymalizować program poprzez zapisywanie i pobieranie wartości z rejestru. Można również deklarować wskaźniki do pamięci `volatile`.

- C++11 oferuje również magiczne zaklęcie `std::atomic<>` sprawiające, że pamięć zachowuje się przez chwilę tak, jakby naprawdę była prostym, liniowym magazynem bajtów, ignorując skomplikowane aspekty nowoczesnych procesorów, takie jak wiele wątków wykonania, wielowarstwowa pamięć podręczna itp. Niektórzy programiści błędnie zakładają, że do tego celu służy słowo `volatile`.

System operacyjny okłamuje również programy i ich użytkowników. Co więcej, przekazywanie programom zbioru bardzo przekonujących kłamstw stanowi w rzeczywistości główny cel systemu operacyjnego. System operacyjny chce przede wszystkim, aby każdy program uwierzył, że tylko on działa na komputerze, że fizyczna pamięć jest nieograniczona oraz że wątki programu mają do dyspozycji nieograniczoną liczbę procesorów.

System operacyjny wykorzystuje warstwę sprzętową komputera do ukrycia tych kłamstw, a zatem programy C++ nie mają innego wyjścia, jak w nie uwierzyć. Zasadniczo te kłamstwa nie mają ogromnego wpływu na działanie programu poza jego spowolnieniem. Jednak mogą one komplikować mierzenie wydajności.

Prawda o komputerach

Tylko najprostsze mikroprocesory i niektóre dawne komputery typu mainframe bezpośrednio odpowiadają modelowi C++. W kontekście optymalizacji ważny jest fakt, iż w rzeczywistości sprzętowa pamięć prawdziwych komputerów jest dużo wolniejsza niż częstotliwość wykonania instrukcji, dostęp do pamięci nie przebiega w bajtach, pamięć nie jest po prostu liniową tablicą identycznych komórek i ma ograniczoną pojemność. Prawdziwe komputery mogą mieć więcej niż jeden adres instrukcji. Prawdziwe komputery są szybkie nie dlatego, że szybko wykonują każdą instrukcję, ale dlatego, że wykonują wiele instrukcji jednocześnie i zawierają skomplikowany mechanizm zapewniający, że nakładające się instrukcje działają tak samo, jak gdyby były wykonywane jedna po drugiej.

Pamięć jest powolna

Główna pamięć komputera działa dużo wolniej niż wewnętrzne bramki i rejestry. Przesłanie elektronów z chipu mikroprocesora we względną otchłań miedzianych ścieżek obwodu, a następnie przepchnięcie ich przez ścieżkę do znajdującego się kilka centymetrów dalej chipu pamięci zajmuje tysiąckrotnie więcej czasu niż przesłanie elektronów między znajdującymi się bardzo blisko siebie tranzystorami mikroprocesora. Główna pamięć jest tak wolna, że komputer może wykonać setki instrukcji w czasie, jaki zajmuje pobranie pojedynczego słowa z głównej pamięci.

Z perspektywy optymalizacji warto zapamiętać, że dostęp do pamięci *przewyższa inne koszty związane z procesorem*, łącznie z kosztem wykonywania instrukcji.

Wąskie gardło architektury von Neumanna

Interfejs do głównej pamięci stanowi punkt ograniczający szybkość wykonania. Ten ograniczający punkt został nazwany *wąskim gardłem architektury von Neumanna*, po słynnym pionierze informatyki i matematyku Johnie von Neumannie (1903–1957).

Na przykład, komputer PC z urządzeniami pamięci DDR2 o częstotliwości 1000 MHz (popularnej kilka lat temu i ułatwiającej dokonywanie obliczeń) ma teoretyczną przepustowość 2 miliardów słów na sekundę lub 500 pikosekund (ps) na słowo. Jednak to wcale nie oznacza, że komputer może odczytywać lub zapisywać losowe słowo danych co 500 pikosekund.

Po pierwsze, tylko sekwencyjny dostęp można zakończyć w jednym cyklu (połowie tyknięcia 1000 MHz zegara). Dostęp do odległej lokalizacji zajmuje zazwyczaj od 6 do 10 cyklów.

Po drugie, różne aktywności rywalizują o dostęp do magistrali pamięci. Procesor stale sięga do pamięci zawierającej następną instrukcję do wykonania. Kontroler pamięci podręcznej pobiera bloki pamięci danych i odsyła zapisane linie pamięci podręcznej. Kontroler DRAM również przywłaszcza pewne cykle, aby odświeżyć zawartość komórek dynamicznej pamięci RAM. Liczba rdzeni w wielordzeniowym procesorze jest wystarczająco wysoka, aby powodować stałe obciążenie magistrali pamięci. Rzeczywiste tempo odczytu danych z głównej pamięci do określonego rdzenia wynosi zwykle około 20–80 nanosekund (ns) na słowo.

Zgodnie z prawem Moore'a każdego roku można umieścić w mikroprocesorze dodatkowe rdzenie. Jednak to nie przyspiesza interfejsu głównej pamięci. W konsekwencji nawet podwojenie liczby rdzeni ma nikły wpływ na wydajność. Przeszkodę będzie stanowił ograniczony dostęp tych rdzeni do pamięci. To nieuniknione ograniczenie wydajności nazywane jest *ścianą pamięci*.

Dostęp do pamięci nie zamyka się w bajtach

Chociaż język C++ wierzy w możliwość uzyskiwania dostępu do każdego pojedynczego bajta, komputery często kompensują wolny dostęp do pamięci fizycznej, pobierając większe porcje danych. Niektóre najmniejsze procesory pobierają z głównej pamięci pojedyncze bajty, jednak procesory powszechnie stosowane w komputerach osobistych pobierają do 64 bajtów naraz, a superkomputery i procesory graficzne jeszcze więcej.

Gdy język C++ pobiera dane typu wielobajtowego, jak `int`, `double` czy wskaźnik, może się zdarzyć, że bajty wchodzące w skład tych danych zawierają się w dwóch słowach pamięci. Taka sytuacja nazywana jest *niewyrównanym dostępem do pamięci*. W kontekście

optymalizacji oznacza to, że *niewyrównany dostęp trwa dwukrotnie dłużej niż gdyby wszystkie bajty znajdowały się w tym samym słowie*, ponieważ trzeba odczytać dwa słowa. Kompilator C++ próbuje wyrównać położenie struktur w taki sposób, aby każde pole rozpoczynało się od adresu bajta stanowiącego wielokrotność rozmiaru pola. Jednak to pociąga za sobą kolejny problem, a mianowicie „dziury" w strukturze zawierające niewykorzystane dane. Zwrócenie uwagi na rozmiar pól danych i ich kolejność w strukturach pomaga w uzyskaniu struktur, które są maksymalnie zwięzłe i odpowiednio wyrównane.

Nie wszystkie operacje dostępu do pamięci są równie wolne

Aby dodatkowo zrekompensować wolne działanie pamięci głównej, wiele komputerów posiada *pamięć podręczną* (nazywaną również pamięcią cache lub buforem). Pamięć podręczna to pewnego rodzaju tymczasowy magazyn umieszczony bardzo blisko procesora, aby przyspieszyć uzyskiwanie dostępu do najczęściej wykorzystywanych słów. Niektóre komputery nie mają pamięci podręcznej, inne mają kilka poziomów pamięci podręcznej, każdy z nich mniejszy, szybszy i kosztowniejszy niż poprzedni. Gdy jednostka wykonania potrzebuje bajtów ze zbuforowanego słowa, może natychmiast je pobrać bez konieczności ponownego uzyskiwania dostępu do pamięci głównej. Jak szybko działa pamięć podręczna? Generalnie zasada jest taka, że każdy poziom pamięci podręcznej jest około 10 razy szybszy niż poziom znajdujący się pod nim w hierarchii pamięci. W przypadku procesorów w komputerach PC czas uzyskiwania dostępu do pamięci może różnić się o nawet pięć rzędów wielkości w zależności od tego, czy dane znajdują się na pierwszym, drugim bądź trzecim poziomie pamięci podręcznej, w pamięci głównej lub na stronie wirtualnej pamięci na dysku. Z tego powodu analizowanie cyklów zegara i innych aspektów wykonywania instrukcji bywa tak frustrujące i niepraktyczne. Stan pamięci podręcznej sprawia, że czasy wykonania instrukcji jawią się jako niedeterministyczne.

Gdy jednostka wykonania potrzebuje danych, które nie znajdują się w pamięci podręcznej, pewne dane zlokalizowane aktualnie w pamięci podręcznej muszą zostać usunięte w celu zwolnienia miejsca. Zazwyczaj usuwane są dane, które od najdłuższego czasu nie były wykorzystywane. Ma to znaczenie w kontekście optymalizacji, ponieważ oznacza, że *dostęp do intensywniej wykorzystywanych lokalizacji w pamięci zajmuje mniej czasu niż dostęp do rzadziej wykorzystywanych lokalizacji.*

Odczytanie choćby jednego bajta danych nieznajdujących się w pamięci podręcznej powoduje zbuforowanie także pobliskich danych (co skutkuje usunięciem z pamięci wielu innych zbuforowanych bajtów). Te pobliskie bajty stają się łatwo dostępne. Ma to znaczenie z punktu widzenia optymalizacji, ponieważ oznacza, że *dostęp do sąsiadujących lokalizacji w pamięci zajmuje (średnio) mniej czasu niż dostęp do odleglejszych lokalizacji.*

W kontekście języka C++ oznacza to, że blok kodu zawierający pętle może być wykonywany szybciej, ponieważ instrukcje tworzące pętle są intensywnie wykorzystywane i położone obok siebie, a zatem z dużym prawdopodobieństwem znajdują się już w pamięci podręcznej. Blok kodu zawierający wywołania funkcji lub instrukcji `if`, których wykonanie wiąże się z przeskokiem, może działać wolniej, ponieważ poszczególne części

kodu są rzadziej wykonywane i nie znajdują się blisko siebie. Tego typu bloki kodu zużywają więcej miejsca w pamięci podręcznej niż ścisła pętla. Jeśli program jest duży i pamięć podręczna ograniczona, część kodu musi zostać usunięta z pamięci w celu zwolnienia miejsca na inne dane, co wydłuża czas uzyskiwania dostępu do pozostałych części kodu. Analogicznie, struktura danych obejmująca kolejne lokalizacje, taka jak tablica lub wektor, może pozwalać na szybszy dostęp niż struktura danych składająca się z węzłów powiązanych wskaźnikami, ponieważ sąsiadujące dane zostają przechowane w niewielkiej liczbie lokalizacji w pamięci podręcznej. Dostęp do struktury rekordów powiązanych wskaźnikami (jak np. lista czy drzewo) może przebiegać wolniej, ponieważ dane z poszczególnych węzłów w pamięci głównej muszą zostać wczytane do pamięci podręcznej.

Słowa mają najbardziej i najmniej znaczący koniec

Możliwe jest pobranie z pamięci pojedynczego bajta danych, ale często pobieranych jest kilka kolejnych bajtów tworzących liczbę. Na przykład, w Visual C++ firmy Microsoft cztery pobierane razem bajty tworzą liczbę typu `int`. Istnieją dwa sposoby wykorzystywania tej samej porcji pamięci, dlatego projektanci komputerów muszą podjąć decyzję. Czy pierwszy bajt, ten o najmniejszym adresie, będzie reprezentować najbardziej znaczące czy najmniej znaczące bity liczby `int`?

Na pierwszy rzut oka może się wydawać, że ta decyzja nie ma znaczenia. O ile oczywiście wszystkie komponenty komputera wiedzą, które bity liczby `int` zostały zapisane na początku. W przeciwnym przypadku zapanowałby absolutny chaos. Różnica jest ewidentna. Gdy liczba `int` o wartości 0x01234567 jest przechowywana pod adresami 1000–1003 i najbardziej znaczące bity są przechowywane na początku, adres 1000 zawiera bajt 0x01, a adres 1003 zawiera bajt 0x67. Natomiast gdy na początku przechowywane są najmniej znaczące bity, adres 1000 zawiera bajt 0x67, a adres 1003 bajt 0x01. Komputery odczytujące najbardziej znaczące bity na początku noszą nazwę *big-endian*, natomiast te odczytujące najpierw najmniej znaczące bity noszą nazwę *little-endian*. Oba sposoby przechowywania liczb (lub wskaźników) są równorzędne, dlatego różne zespoły projektujące różne procesory dla różnych firm mogą dokonać innego wyboru.

Problem pojawia się wtedy, gdy dane zapisane na dysku lub przesłane za pośrednictwem sieci przez jeden komputer muszą zostać odczytane przez inny komputer. Dyski i sieci przesyłają pojedyncze bajty, nie całe wartości `int`. W związku z tym sposób zapisywania lub przesyłania liczb ma znaczenie. Jeśli przesyłający i odbierający komputer nie są zgodne, wartość przesłana jako 0x01234567 zostanie odebrana jako zupełnie inna wartość 0x67452301.

Różnica w sposobie zapisywania najbardziej znaczących bajtów przez różne komputery stanowi jeden z powodów, dla jakich język C++ nie może określać sposobu rozmieszczania bitów w wartości `int` oraz dlaczego ustawienie jednego pola w unii wpływa na pozostałe pola. Jest to jedna z przyczyn tego, że pewne programy działają prawidłowo tylko na niektórych komputerach.

Pamięć ma ograniczoną pojemność

W rzeczywistości komputer nie posiada nieograniczonej pamięci. Aby zachować iluzję nieograniczonej pamięci, system operacyjny może wykorzystywać pamięć fizyczną w roli pamięci podręcznej i zapisywać dane niemieszczące się w pamięci fizycznej w postaci pliku na dysku. Takie rozwiązanie nazywane jest *pamięcią wirtualną*. Pamięć wirtualna stwarza iluzję istnienia dodatkowej pamięci fizycznej. Jednak pobieranie bloku pamięci z dysku zabiera kilkadziesiąt milisekund, czyli w kontekście nowoczesnych komputerów niemal wieczność.

Zapewnianie szybkości pamięci podręcznej jest kosztowne. Komputer lub smartfon może zawierać gigabajty pamięci głównej, ale tylko kilka milionów bajtów pamięci podręcznej. Programy i ich dane nie mieszczą się zwykle w pamięci podręcznej.

Jedną z konsekwencji stosowania pamięci podręcznej i wirtualnej jest to, że *w wyniku buforowania funkcja uruchomiona w kontekście całego programu może działać wolniej niż ta sama funkcja uruchomiona w warunkach testowych*, gdy zostaje uruchomiona 10000 razy w celu dokonania pomiaru wydajności. W kontekście wykonania całego programu funkcja i jej dane nie znajdują się prawdopodobnie w pamięci podręcznej, natomiast w warunkach testowych można się ich tam spodziewać. Ten efekt wyolbrzymia korzyści optymalizacji redukującej zużycie pamięci lub dysku, nie ma natomiast wpływu na optymalizację redukującą rozmiar kodu.

Buforowanie powoduje również, że gdy duży program dokonuje wielu odczytów różnych lokalizacji w pamięci, pojemność pamięci podręcznej może nie wystarczyć do przechowania danych bezpośrednio wykorzystywanych przez program. To prowadzi do wystąpienia niekorzystnego zjawiska nazywanego *migotaniem stron* (ang. *page thrashing*). Gdy migotanie stron występuje w wewnętrznej pamięci podręcznej mikroprocesora, konsekwencją jest obniżenie wydajności. Gdy występuje ono w pliku pamięci wirtualnej systemu operacyjnego, wydajność obniża się tysiąckrotnie. Ten problem pojawiał się częściej, gdy komputery miały mniej pamięci fizycznej, ale nadal występuje.

Wykonanie instrukcji zabiera dużo czasu

Proste mikroprocesory, takie jakie są wbudowywane w ekspresy do kawy lub kuchenki mikrofalowe, zostały zaprojektowane tak, aby wykonywać instrukcje tak szybko, jak tylko zostaną one pobrane z pamięci. Mikroprocesory w komputerach stacjonarnych posiadają dodatkowe zasoby do równoczesnego przetwarzania wielu instrukcji i mogą wykonywać instrukcje wielokrotnie szybciej niż trwa pobranie ich z pamięci głównej, z reguły polegając na szybkiej pamięci podręcznej zaopatrującej jednostki wykonania. Taka optymalizacja oznacza, że *dostęp do pamięci przewyższa koszt obliczeń*.

Nowoczesne komputery stacjonarne wykonują instrukcje w niesamowitym tempie, *jeśli* nic im nie przeszkadza. Mogą kończyć instrukcję co kilkaset pikosekund (pikosekunda to 10^{-12} sekundy, niewyobrażalnie krótki czas). To jednak nie oznacza, że wykonanie każdej instrukcji trwa jedynie pikosekundy. Procesor zawiera „potok" instrukcji, nad

którymi stale pracuje. Instrukcje są przekazywane w potoku i po drodze dekodowane, ich argumenty są pobierane, ich obliczenia wykonywane, ich wyniki zapisywane. Im bardziej zaawansowany procesor, tym bardziej skomplikowany staje się ten potok. Proces wykonania instrukcji może zostać podzielony na kilkanaście faz, aby umożliwić równoczesne przetwarzanie większej liczby instrukcji.

Jeśli instrukcja A wylicza wartość, której potrzebuje instrukcja B, instrukcja B nie może dokonać obliczeń do momentu, aż instrukcja A zwróci wynik. To powoduje *zatrzymanie potoku*, czyli krótką przerwę w procesie wykonywania instrukcji wynikającą z faktu, iż dwie instrukcje nie mogą się całkowicie nakładać. Zatrzymanie potoku trwa szczególnie długo, gdy instrukcja A pobiera wartość z pamięci, a następnie wykonuje obliczenia, których wynik jest potrzebny instrukcji B. Zatrzymania potoku niweczą korzyści płynące z zaawansowanej technologii mikroprocesora, czasami czyniąc go równie powolnym co procesor w tosterze.

Komputery mają trudności z podejmowaniem decyzji

Zatrzymania potoku mogą mieć miejsce również wtedy, gdy komputer musi podjąć decyzję. W przypadku większości instrukcji po ich zakończeniu wykonywana jest instrukcja o kolejnej pozycji w pamięci. Z reguły ta kolejna instrukcja znajduje się już w pamięci podręcznej. Dalsze instrukcje mogą zostać przekazane do potoku, gdy tylko pierwsza faza potoku staje się dostępna.

Instrukcje zmieniające przepływ sterowania działają inaczej. Instrukcja skoku lub skoku do procedury zmienia adres wykonania na dowolną nową wartość. „Następna" instrukcja nie może zostać odczytana z pamięci i umieszczona w potoku, dopóki w procesie przetwarzania instrukcji skoku nie nastąpi aktualizacja adresu wykonania. Słowo zapisane pod nowym adresem wykonania często nie znajduje się jeszcze w pamięci podręcznej. Potok zostaje zatrzymany do momentu zaktualizowania adresu wykonania i załadowania do potoku nowej „następnej" instrukcji.

Natomiast po warunkowej instrukcji rozgałęzienia wykonanie jest kontynuowane w jednym z dwóch różnych miejsc, w zależności od wyniku wcześniejszego obliczenia: kolejnej instrukcji lub instrukcji „else" znajdującej się pod adresem rozgałęzienia. Potok zostaje zatrzymany do momentu zakończenia wszystkich instrukcji wchodzących w skład wcześniejszego obliczenia i pozostaje zatrzymany do czasu ustalenia nowego adresu wykonania i pobrania wartości znajdującej się pod tym adresem.

Z perspektywy optymalizacji oznacza to, że *obliczanie jest szybsze niż podejmowanie decyzji*.

Istnieje wiele strumieni wykonania programu

Każdy program uruchomiony w nowoczesnym systemie operacyjnym współdzieli komputer z innymi działającymi jednocześnie programami, procesami konserwacyjnymi przeprowadzającymi okresowe sprawdzanie dysku bądź wyszukującymi aktualizacje Java lub

Flash, a także różnymi składnikami systemu operacyjnego kontrolującymi interfejs sieciowy, dyski, urządzenia dźwiękowe, akceleratory, termometry i inne urządzenia zewnętrzne. *Każdy program rywalizuje z innymi programami o zasoby komputera.*

Z reguły program za bardzo tego nie odczuwa, po prostu działa nieco wolniej. Za wyjątkiem sytuacji, gdy wiele programów jest uruchamianych jednocześnie i wszystkie rywalizują o pamięć oraz dysk. Z perspektywy dostosowywania wydajności: *jeśli program ma działać w czasie uruchomienia lub maksymalnego obciążenia, pomiary wydajności muszą być dokonywane pod obciążeniem*.

Na początku 2016 roku sytuacja wyglądała następująco: komputery stacjonarne zawierały do 16 rdzeni procesora, mikroprocesory stosowane w telefonach i tabletach do 8 rdzeni. Wystarczy zajrzeć do Menedżera zadań systemu Windows, listy stanu procesów w systemie Linux lub listy zadań w systemie Android, aby zauważyć, że zazwyczaj uruchomionych jest dużo więcej procesów oprogramowania, a większość procesów posiada wiele wątków wykonania. System operacyjny wykonuje wątek przez krótki czas, a następnie przełącza kontekst na inny wątek lub proces. Z perspektywy programu wygląda to tak, jakby wykonanie jednej instrukcji zajęło nanosekundę, a kolejnej 60 milisekund.

Co oznacza przełączenie kontekstu? Jeśli system operacyjny przełącza się między różnymi wątkami tego samego programu, oznacza ono zapisanie rejestrów procesora dla zawieszanego wątku i załadowanie zapisanych rejestrów dla wznawianego wątku. Rejestry nowoczesnych procesorów zawierają setki bajtów danych. Gdy nowy wątek wznawia wykonanie, jego dane mogą nie znajdować się w pamięci podręcznej, a zatem może nastąpić pewien okres wolnego wykonania, w czasie gdy nowy kontekst jest ładowany do pamięci podręcznej. W związku z tym przełączanie się między kontekstami wątków stanowi kosztowną operację.

Gdy system operacyjny przełącza kontekst z jednego programu na drugi, koszt jest jeszcze większy. Wszystkie zanieczyszczone strony pamięci podręcznej (zawierające zapisane dane, które nie dotarły jeszcze do pamięci głównej) muszą zostać zrzucone do pamięci fizycznej. Wszystkie rejestry procesora muszą zostać zapisane. Później zapisywane są rejestry stron pamięci fizycznej do wirtualnej z menedżera pamięci. Następnie ładowane są rejestry stron pamięci fizycznej do wirtualnej oraz rejestry procesora dla nowego procesu. I dopiero wtedy można wznowić wykonanie. Jednak pamięć podręczna jest pusta, dlatego procesowi jej wypełniania towarzyszy początkowy okres obniżonej wydajności i wysokiej rywalizacji o zasoby.

Gdy program musi oczekiwać na wystąpienie określonego zdarzenia, czasami musi dodatkowo oczekiwać, aż system operacyjny udostępni procesor w celu kontynuowania programu. W efekcie program może działać dłużej bądź w mniej przewidywalny sposób, gdy jest uruchamiany wraz z innymi programami rywalizującymi o zasoby komputera.

Jednostki wykonania wielordzeniowego procesora i powiązane z nimi pamięci podręczne działają w dużym stopniu niezależnie od siebie w celu osiągnięcia wyższej wydajności. Jednak wszystkie jednostki wykonania korzystają z tej samej pamięci głównej. Jednostki wykonania muszą rywalizować o dostęp do sprzętu łączącego je z pamięcią

główną, co sprawia, że wąskie gardło architektury von Neumanna pociąga za sobą jeszcze większe ograniczenia na komputerze z wieloma jednostkami wykonania.

Gdy jednostka wykonania zapisuje wartość, wartość jest najpierw przekazywana do pamięci podręcznej. Aczkolwiek prędzej czy później będzie ona musiała zostać zapisana w pamięci głównej, aby stała się dostępna także dla pozostałych jednostek wykonania. Jednak ponieważ jednostki wykonania rywalizują o dostęp do pamięci głównej, uaktualnienie pamięci głównej może nastąpić setki instrukcji po zmodyfikowaniu wartości przez jednostkę wykonania.

Jeśli komputer ma wiele jednostek wykonania, jedna jednostka wykonania może długo nie widzieć modyfikacji danych zapisywanej przez inną jednostkę wykonania w pamięci głównej i w konsekwencji zmiany w pamięci głównej mogą zachodzić w kolejności niezgodnej z kolejnością wykonania instrukcji. W zależności od nieprzewidywalnych czynników czasowych jednostka wykonania może zobaczyć albo starą wartość współdzielonego słowa w pamięci, albo zmodyfikowaną wartość. Aby zapewnić wątkom uruchomionym w różnych jednostkach wykonania spójny wgląd w pamięć, trzeba użyć specjalnych instrukcji synchronizacji. Z perspektywy optymalizacji oznacza to, że *dostęp do danych współdzielonych przez różne wątki wykonania zajmuje dużo więcej czasu niż dostęp do niewspółdzielonych danych*.

Wywoływanie systemu operacyjnego jest kosztowne

Wszystkie procesory, poza tymi najmniejszymi, zawierają rozwiązania sprzętowe służące do wymuszania izolacji między programami, aby program A nie mógł dokonywać odczytów ani zapisów w pamięci fizycznej należącej do programu B. To samo rozwiązanie sprzętowe chroni jądro systemu operacyjnego przed nadpisaniem przez inne programy. Natomiast jądro systemu operacyjnego musi mieć dostęp do pamięci należącej do wszystkich programów, aby programy mogły realizować wywołania systemowe. Niektóre systemy operacyjne zezwalają również programom na współdzielenie pamięci. Sposoby realizowania wywołań systemowych i współdzielenia pamięci są różne i skomplikowane. Z perspektywy optymalizacji *wywołania systemowe są kosztowne*, kilkusetkrotnie kosztowniejsze niż wywoływanie funkcji w ramach pojedynczego wątku programu.

C++ również kłamie

Największym kłamstwem, jakie język C++ mówi użytkownikom, jest to, że komputer ma prostą, spójną strukturę. Ponieważ programiści udają, że wierzą w to kłamstwo, język C++ pozwala im programować bez konieczności szczegółowego zapoznawania się z każdym mikroprocesorem, w odróżnieniu od programowania w brutalnie szczerym języku asemblera.

Różne instrukcje mają różny koszt

W dawno minionych, spokojnych czasach programowania w języku C według Ritchiego i Kernighana wszystkie instrukcje miały w zasadzie taki sam koszt. Wywołanie funkcji mogło zawierać dowolnie złożone obliczenia. Jednak instrukcja przypisania z reguły kopiowała coś mieszczącego się w rejestrze maszyny do czegoś innego, również mieszczącego się w rejestrze maszyny. A zatem instrukcja:

```
int i,j;
 ...
i = j;
```

kopiowała 2 lub 4 bajty z `j` do `i`. Niezależnie od tego, czy zadeklarowany został typ `int`, `float` czy `struct bigstruct *`, instrukcja przypisywania wykonywała podobną pracę.

Jednak sytuacja ta uległa zmianie. W języku C++ przypisanie jednej zmiennej `int` wartości innej zmiennej `int` zabiera tyle samo pracy, ile analogiczna instrukcja C. Jednak instrukcja typu `BigInstance i = OtherObject;` może kopiować całe struktury. A co ważniejsze, tego typu przypisanie wiąże się z wywołaniem funkcji konstruktora `BigInstance`, za którym może się kryć dowolnie skomplikowana maszyneria. Konstruktor jest również wywoływany dla każdego wyrażenia przekazywanego do formalnego argumentu funkcji oraz ponownie, gdy funkcja zwraca wartość. Operatory arytmetyczne i porównania również mogą być przeładowywane, a zatem `A=B*C;` może służyć do mnożenia *n*-wymiarowych macierzy, a `if(x<y)...` może wymagać porównania dwóch ścieżek w dowolnie złożonym grafie skierowanym. Z perspektywy optymalizacji oznacza to, że *niektóre instrukcje zawierają wysoką liczbę obliczeń. Postać instrukcji nie wskazuje, jaki jest jej koszt.*

Ta informacja prawdopodobnie nie będzie zaskoczeniem dla programistów, którzy rozpoczęli naukę od języka C++. Jednak instynkt tych programistów, którzy najpierw poznali C, może wywodzić ich na manowce.

Instrukcje nie są wykonywane kolejno

Programy C++ zachowują się w taki sposób, jakby instrukcje były wykonywane w oryginalnej kolejności (z uwzględnieniem instrukcji przepływu sterowania C++). Sprytne sformułowanie „jakby” użyte w poprzednim zdaniu sygnalizuje, że optymalizatory dostępne w kompilatorach oraz nowoczesny sprzęt komputerowy mogą stosować pewne sztuczki.

W rzeczywistości kompilator może zmieniać kolejność instrukcji, co często czyni w celu poprawienia wydajności. Jednocześnie kompilator ma świadomość, że zmienna musi zawierać aktualny wynik przypisanego jej obliczenia, zanim zostanie przetestowana lub przypisana innej zmiennej. Nowoczesne mikroprocesory mogą również zdecydować się na wykonywanie instrukcji w zmienionej kolejności, pilnując jednocześnie, aby zapisy w pamięci następowały przed dalszymi odczytami tej lokalizacji. Wprawdzie funkcja mikroprocesora kontrolująca zapisy w pamięci może opóźniać je, aby w optymalny sposób wykorzystać magistralę pamięci, ale kontroler pamięci wie, które zapisy są aktualnie